Алгоритмические трюки для программистов

ИСПРАВЛЕННОЕ ИЗДАНИЕ

Hacker's Delight

Henry S. Warren, Jr.



ADDISON-WESLEY

Boston ♦ San Francisco * New York ♦ Toronto 4 Montreal
London ♦ Munich * Paris 4 Madrid
Capetown ♦ Sydney ♦ Tokyo ♦ Singapore ♦ Mexico City

Алгоритмические трюки для программистов

ИСПРАВЛЕННОЕ ИЗДАНИЕ

Генри Уоррен, мл.



Издательский дом "Вильяме"
Москва ♦ Санкт-Петербург * Кисв
2004

ББК 32.973.26-018.2.75
У64
УДК 681.3.07

Издательский дом "Вильяме"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского канд. техн наук *И.В. Красикова, Г.И. Сингаевской*

Под редакцией канд. техн наук *И.В. Красикова*

По общим вопросам обращайтесь в Издательский дом "Вильяме" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Уоррен. Генри, С.

У64 Алгоритмические трюки для программистов, испр. изд. : Пер. с англ. — М. : Издательский дом "Вильяме", 2004. — 288с. : ил. — Парал. тит. англ.

ISBN 5-8459-0572-9 (рус.)

В этой книге слову "хакер" возвращено его первозданное значение — человека увлеченного, талантливого программиста, способного к созданию чрезвычайно эффективного и элегантного кода. В книге воплощен сорокалетний стаж ее автора в области разработки компиляторов и архитектуры компьютеров. Здесь вы найдете множество приемов для работы с отдельными битами, байтами, вычисления различных целочисленных функций; большей части материала сопутствует строгое математическое обоснование. Каким бы ни был ваш уровень профессиональной подготовки — вы обязательно найдете в этой книге новое для себя; кроме того, книга заставит вас посмотреть на уже знакомые вещи с новой стороны. Не в меньшей степени эта книга пригодится и начинающему программисту, который может просто воспользоваться готовыми советами из книги, применяя их в своей повседневной практике.

ББК 32.973.26-018.2.75

ISBN 5-8459-0572-9 (рус.)
ISBN 0-201-91465-4 (англ.)

ОГЛАВЛЕНИЕ

СОДЕРЖАНИЕ

*Джозефу У. Гауду (Joseph W. Gaud),
моему школьному учителю алгебры,
который зажег во мне пламя
восхищения математикой*

ПРЕДИСЛОВИЕ

Около 30 лет назад, во время первой летней практики, мне посчастливилось участвовать в проекте МАС в Массачусетском технологическом институте. Тогда я впервые смог поработать на компьютере DEC PDP-10, который в то время предоставлял более широкие возможности при программировании на ассемблере, чем любой другой компьютер. Этот процессор имел большой набор команд для выполнения операций с отдельными битами, их тестирования, маскирования, для работы с полями и целыми числами. Несмотря на то что PDP-10 давно снят с производства, до сих пор есть энтузиасты, работающие на этих машинах либо на их программных эмуляторах. Некоторые из них даже пишут новые приложения; по крайней мере в настоящее время есть, как минимум, один Web-узел, поддерживаемый эмулированным PDP-10. (Не смейтесь — поддержка такого узла ни в чем не уступает поддержке антикварного автомобиля "на ходу".)

Тем же летом 1972 года я увлекся чтением серии отчетов по новым исследованиям, публиковавшихся в институте под общим названием НАКМЕМ — довольно странный и эклектичный сборник разнообразных технических мелочей[1]. Тематика статей могла быть любой: от электронных схем до теории чисел; но больше всего меня заинтриговал небольшой раздел, в котором публиковались программистские хитрости. Практически в каждой статье этого раздела содержалось описание некоторой (зачастую необычной) операции над целыми числами или битовыми строками (например, подсчет единичных битов в слове), которую можно было легко реализовать в виде длинной последовательности машинных команд либо цикла, а затем обсуждалось, как можно сделать это же, используя только четыре, три или две тщательно подобранные команды, взаимодействие между которыми становилось очевидным только после соответствующих объяснений или самостоятельных исследований. Я изучал эти маленькие программные самородки с тем же наслаждением, с которым другие пьют пиво или едят сладости. Я просто не мог остановиться. Каждая такая программа оказывалась для меня находкой, в каждой из них была интеллектуальная глубина, элегантность и даже своеобразная поэзия.

"Наверняка, — думал я тогда, — таких программ должно быть намного больше. Должна же где-то быть книга, посвященная таким штукам".

Книга, которую вы держите в руках, потрясла меня. Автор систематически, на протяжении многих лет собирал такие программные перлы, а теперь свел их воедино, тематически организовал и снабдил каждый четким и подробным описанием. Многие из них можно записать в машинных командах, но книга полезна не только для тех, кто пишет программы на ассемблере. Главная тема книги — рассмотреть базовые структурные отношения среди целых чисел и битовых строк и эффективные приемы реализации операций над ними. Рассмотренные в книге методы столь же полезны и практичны при программировании на языках высокого уровня, например С или Java, как и на языке ассемблера.

Во многих книгах по алгоритмам и структурам данных описаны сложные методы сортировки и поиска, поддержания хэш-таблиц и двоичных деревьев, работы с записями и указателями. Здесь же рассматривается, что можно сделать с очень крошечной частью

---

[1] Почему НАКМЕМ? Сокращение от hacks memo; одно 36-битовое PDP-10 слово может содержать шесть 6-битовых символов, поэтому многие имена PDP-10, с которыми тогда работали программисты, были ограничены шестью символами. Аббревиатуры из шести символов широко использовались для сокращенных названий, иногда ими пользовались просто для удобства Поэтому название сборника НАКМЕМ имело ясный смысл для специалистов, по крайней мере в то время.

данных — битами и массивами битов. Поразительно, сколько всего можно сделать, используя только операции двоичного сложения и вычитания вместе с некоторыми поразрядными операциями. Операции с переносом позволяют одному биту воздействовать на все биты, находящиеся слева от него, что делает сложение особенно мощной операцией при работе с данными способами, не получившими широкого распространения.

Сейчас у вас в руках книга именно о таких методах. Если вы работаете над оптимизирующим компилятором или просто создаете высокопроизводительный код, вам просто необходимо прочитать эту книгу. Возможно, в своей повседневной работе вы не часто будете использовать изложенные здесь приемы, но, если вам потребуется организовать цикл по всем битам слова, ускорить работу с отдельными битами во внутреннем цикле или вы просто столкнетесь с ситуацией, когда код получается слишком сложным, загляните в эту книгу — я уверен, вы найдете в ней помощь в решении стоящей перед вами проблемы. В любом случае чтение этой книги доставит вам огромное удовольствие.

Гай Л. Стал, мл. (Guy L. Steele, Jr.)
Берлингтон, Массачусетс
Апрель 2002

ВСТУПЛЕНИЕ

Caveat emptor[2]*: стоимость сопровождения программного обеспечения пропорциональна квадрату творческих способностей программиста.*

(Первый закон творческого программирования, Роберт Д. Блис (Robert D. Bliss), 1992)

Перед вами сборник программных приемов, которые я собирал много лет. Большинство из них работают только на компьютерах, в которых целые числа представлены в дополнительном коде. Хотя в данной книге речь идет о 32-битовых машинах с соответствующей длиной регистра, большую часть представленных здесь алгоритмов легко перенести на машины с другими размерами регистров.

В этой книге не рассматриваются сложные вопросы наподобие методов сортировки или оптимизации компилируемого кода. Основное внимание уделено приемам работы с отдельными машинными словами или командами, например подсчету количества единичных битов в заданном слове. В подобных приемах часто используется смесь арифметических и логических команд.

Предполагается, что прерывания, связанные с переполнением целых чисел, замаскированы и произойти не могут. Программы на C, Fortran и даже Java работают в таком окружении, но программистам на Pascal и ADA следует быть осторожными!

Представление материала в книге неформальное. Доказательства приводятся только в том случае, если алгоритм неочевиден, а иногда не приводятся вообще. Все методы используют компьютерную арифметику, базовые функции, комбинации арифметических и логических операций и др., а доказательства теорем в этой предметной области часто сложны и громоздки.

Чтобы свести к минимуму количество типографских ошибок и опечаток, многие алгоритмы реализованы на языке программирования высокого уровня, в качестве которого используется C. Это обусловлено его распространенностью и тем, что он позволяет непосредственно комбинировать операции с целыми числами и битовыми строками; кроме того, компилятор языка C генерирует объектный код высокого качества.

Ряд алгоритмов написан на машинном языке. В книге применяется трехадресный формат команд, главным образом для повышения удобочитаемости. Использован язык ассемблера для некой абстрактной машины, которая является представителем современных RISC-компьютеров.

Отсутствие ветвлений в программе всячески приветствуется. Это связано с тем, что на многих машинах наличие ветвлений замедляет выборку команд и блокирует их параллельное выполнение. Кроме того, наличие ветвлений может препятствовать выполнению оптимизации компилятором. Оптимизирующий компилятор более эффективно работает с несколькими большими блоками кода, чем с множеством небольших.

Крайне желательно использовать малые величины при непосредственном задании операнда, сравнение с 0 (а не с другими числами) и параллелизм на уровне команд. Хотя программу

[2] Caveat emptor (лат.) - да будет осмотрителен покупатель.

часто можно значительно сократить за счет использования поиска в таблице, этот метод не слишком распространен. Дело в том, что по сравнению с выполнением арифметических команд загрузка данных из памяти занимает намного больше времени, а методы поиска в таблице зачастую не представляют большого интереса (хотя и весьма практичны).

Напоследок мне хотелось бы напомнить исходное значение слова хакер[3]. Хакер — это страстный любитель компьютеров, он создает что-то новое, переделывает или совершенствует то, что уже есть. Хакер очень хорошо разбирается в том, что делает, хотя часто не является профессиональным программистом или разработчиком. Обычно хакер пишет программы не ради выгоды, а ради собственного удовольствия. Такая программа может оказаться полезной, а может остаться всего лишь игрой интеллекта. Например, хакер может написать программу, которая во время выполнения выводит точную копию себя самой[4]. Таких людей называют хакерами, и эта книга написана именно для них, а не для тех, кто хочет получить совет о том, как взломать что-либо в компьютере.

Благодарности

Прежде всего я хочу поблагодарить Брюса Шрайвера (Bruce Shriver) и Денниса Аллисона (Dennis Allison), оказавших мне помощь в опубликовании книги. Я признателен коллегам из ЮМ, многие из которых упомянуты в библиографии. Особой благодарности достоин Мартин Хопкинс (Martin E. Hopkins) из IBM, которого по праву можно назвать "мистер Компьютер". Этот человек неудержим в своем стремлении подсчитать в программе каждый такт, и многие его идеи нашли отражение в этой книге. Благодарю также обозревателей из Addison-Wesley, которые значительно улучшили мою книгу. Со многими из них я не знаком, но не могу не упомянуть выдающийся 50-страничный обзор одного из них, Гая Л. Стила, мл. (Guy L. Steele, Jr.), где, в частности, затронуты вопросы перемешивания битов, обобщенного упорядочения и многие другие, которые обязательно войдут во второе издание книги (©). Ряд предложенных им алгоритмов использован в данной книге. Помогла мне и его пунктуальность. Например, я ошибочно написал, что шестнадцатеричное число 0xAAAAAAAA можно разложить на множители 2· 3· 17· 257· 65537; Гай указал, что 3 необходимо заменить на 5. Он не упустил ни одной из подобных мелочей и намного улучшил стиль изложения материала. Кроме того, весь материал, связанный с методом "параллельного префикса", появился в книге исключительно благодаря Гаю. Дополнительные материалы к данной книге можно найти на Web-узле по адресу `www.HackersDelight.org`.

Г.С. Уоррен мл. (H.S. Warren, Jr.)
Йорктаун, Нью-Йорк
Февраль 2002

[3] В последнее время "хакерами" часто называют тех, кто получает несанкционированный доступ к банковским системам, взламывает Web-узлы и ведет прочую разрушительную деятельность либо ради получения денег, либо для демонстрации всем своей "крутости". К настоящим хакерам таковые не имеют никакого отношения. — *Прим. перев.*

[4] Самая короткая такая программа на С, известная автору, написана Владом Таировым и Рашидом Фахреевым и содержит всего 64 символа:

```
main(a){printf(a,34,a="main(a){printf(a,34,a=%c%s%c,34);}",34);}
```

ГЛАВА 1

ВВЕДЕНИЕ

1.1. Система обозначений

Повсюду в книге при описании машинных команд используются выражения, отличающиеся от обычных арифметических выражений. В "компьютерной арифметике" операнды представляют собой битовые строки (или векторы) некоторой фиксированной длины. Выражения компьютерной арифметики похожи на выражения обычной арифметики, но в этом случае переменные обозначают содержимое регистров компьютера. Значение выражения компьютерной арифметики представляет собой строку битов без какой-либо специальной интерпретации; операнды же могут интерпретироваться по-разному. Например, в команде сравнения операнды интерпретируются либо как двоичные целые числа со знаком, либо как беззнаковые целые числа. Поэтому, в зависимости от типа операндов, для обозначения оператора сравнения используются разные символы.

Основное отличие между обычными арифметическими действиями и компьютерными сложением, вычитанием и умножением состоит в том, что результат действий компьютерной арифметики приводится по модулю 2^n, где n — размер машинного слова. Еще одно отличие состоит в том, что компьютерная арифметика содержит большее количество операций. Кроме основных четырех арифметических действий, машина способна выполнить множество других команд: логические *и (and), исключающее или, сравнение, сдвиг влево* и др.

Если не оговорено иное, везде в книге предполагается, что длина слова равна 32 битам, а целые числа со знаком представляются в дополнительном к 2 коде. Если длина слова иная, такие случаи оговариваются особо.

Все выражения компьютерной арифметики записываются аналогично обычным арифметическим выражениям, с тем отличием, что переменные, обозначающие содержимое регистров компьютера, выделяются полужирным шрифтом (это обычное соглашение, принятое в векторной алгебре). Машинное слово интерпретируется как вектор, состоящий из отдельных битов. Константы также выделяются полужирным шрифтом, если обозначают содержимое регистра (в векторной алгебре аналогичного обозначения нет, так как там для записи констант используется только один способ — указание компонентов вектора). Если константа означает часть команды (например, непосредственно значение в команде сдвига), то она не выделяется.

Если оператор, например "+", складывает выделенные полужирным шрифтом операнды, то он подразумевает машинную операцию сложения ("векторное сложение"). Если операнды не выделены, то подразумевается обычное арифметическое сложение скалярных величин (скалярное сложение). Переменная x обозначает арифметическое значение выделенной полужирным шрифтом переменной $\mathbf{x}$ (интерпретируемое как знаковое или беззнаковое, что должно быть понятно из контекста). Таким образом, если $\mathbf{x} = \mathbf{0x80000000}$, а $\mathbf{y} = \mathbf{0x80000000}$, то при знаковой интерпретации $x = y = -2^{31}$, $x + y = -2^{32}$ и $\mathbf{x} + \mathbf{y} = \mathbf{0}$. (Здесь **0x80000000** — шестнадцатеричная запись строки битов, состоящей из одного единичного бита и следующих за ним 31 нулевого бита.)

Биты нумеруются справа налево, причем крайний справа (младший) бит имеет номер 0. Длины бита, полубайта, байта, полуслова, слова и двойного слова равны соответственно 1, 4, 8, 16, 32 и 64 бит.

Кроме функций "abs", "rem" и прочих, в книге используется множество других функций, которые будут определены позже.

При вычислении выражения x < y < z в языке C сначала вычисляется выражение x < y, (результат равен 1, если выражение истинно, и 0, если выражение ложно), затем полученный результат сравнивается с z. Выражение $x<y<z$ с операторами сравнения "<" вычисляется как $(x<y)\&(y<z)$.

В языке C есть три оператора цикла: while, do и for. Цикл while имеет вид:

while *(выражение) оператор*

Перед выполнением цикла вычисляется *выражение.* Если *выражение* истинно (не нуль), выполняется *оператор.* Затем снова вычисляется *выражение.* Цикл while завершается, когда *выражение* становится ложным (равным нулю).

Цикл do аналогичен циклу while, однако проверочное условие стоит после оператора цикла, а именно:

do *оператор* while *(выражение)*

В этом цикле сначала выполняется *оператор,* затем вычисляется *выражение.* Если выражение истинно, операторы цикла выполняются еще раз, если выражение ложно, цикл завершается.

Оператор for имеет вид

for $(e_1;e_2;e_3)$ *оператор*

Сначала вычисляется выражение e_1 — как правило, это оператор присваивания (аналог выражения-инициализации). Затем вычисляется e_2 — оператор сравнения (или условное выражение). Если значение условного выражения равно нулю (условие ложно), цикл завершается, если не равно нулю (условие истинно), выполняется *оператор.* Затем вычисляется e_3 (тоже, как правило, оператор присваивания), и вновь вычисляется условное вьфажение e_2. Таким образом, знакомый всем цикл "do i=1 to n" запишется как for(i=1; i<=n; i++) (это один из контекстов использования оператора постинкремента).

1.2. Система команд и модель оценки времени выполнения команд

Чтобы можно было хотя бы грубо сравнивать алгоритмы, представим, что они кодируются для работы на машине с набором команд, подобных современным RISC-компьютерам общего назначения (типа Compaq Alpha, SGI MIPS и IBM RS/6000). Это трехадресная машина, имеющая достаточно большое количество регистров общего назначения — не менее 16. Если не оговорено иное, все регистры 32-разрядные. Регистр общего назначения с номером 0 всегда содержит нули, все другие регистры равноправны и могут использоваться для любых целей.

Для простоты будем считать, что в компьютере нет регистров "специального назначения", в частности, слова состояния процессора или регистра с битами состояний, например "переполнение". Не рассматриваются также команды для работы с числами с плавающей точкой, как выходящие за рамки тематики данной книги.

В книге описаны два типа RISC: "базовый RISC", команды которого перечислены в табл. 1.2, и "RISC с полным набором команд", в который кроме основных RISC-команд входят дополнительные команды, перечисленные в табл. 1.3.

Таблица 1.2. Базовый набор RISC-команд

Мнемокод команды	**Операнды**	**Описание команды**
add, sub, mul, div, divu, rem, remu	RT, RA, RB	RT←RA op RB Здесь op — сложение *(add)*, вычитание *(sub)*, умножение *(mul)*, знаковое деление *(div)*, беззнаковое деление *(divu)*, остаток от знакового деления *(rem)* или остаток от беззнакового деления*(remu)*
addi, muli	RT, RA, I	RT←RA op I Здесь op — сложение *(addi)* или умножение *(muli)*, I — непосредственно заданное 16-битовое знаковое значение
addis	RT, RA, I	RT←RA+(I || 0X0000)
and, or, xor	RT, RA, RB	RT←RA op RB Здесь op — побитовое *и (and), или (or)* или *исключающее или (xor)*
andi, ori, xori	RT, RA, Iu	То же самое, но последний операнд является непосредственно заданным 16-битовым беззнаковым значением
beq, bne, blt, ble, bgt, bge	RT,target	Переход к метке target, если выполнено некоторое условие, т.е. если RT=0, RT≠0, RT<0, RT≤0, RT>0 или RT≥0 соответственно (RT — целое знаковое число)
bt, bf	RT, target	Переход к метке target, если выполнено некоторое условие (true/false); аналогичны командам bne /beq соответственно
cmpeq, cmpne, cmplt, cmple, cmpgt, cmpge, cmpltu, cmpleu, cmpgtu, cmpgeu	RT, RA, RB	RT содержит результат сравнения RA и RB; RT равен 0, если условие ложно, и 1, если условие истинно. Мнемокоды означают: сравнить на равенство, неравенство, меньше, не больше и т.д, как и в командах условного перехода. Суффикс "и" в названии обозначает сравнение беззнаковых величин
cmpieq, cmpine, cmpilt, cmpile, cmpigt, cmpige	RT, RA, *I*	Аналогичны командам серии cmpeq, но второй операнд представляет собой непосредственно заданное 16-битовоезнаковое значение
cmpiequ, cmpineu, cmpiltu, cmpileu, cmpigtu, cmpigeu	RT, RA, Iu	Аналогичны командам серии cmpltu, но второй операнд представляет собой непосредственно заданное 16-битовоебеззнаковое значение
ldbu, ldh, ldhu, ldw	RT,d(RA)	Загрузка беззнакового байта, знакового полуслова, беззнакового полуслова или слова в RT из ячейки памяти по адресу RA + d, где d — непосредственно заданное знаковое 16-битовое значение

Мнемокод команды	**Операнды**	**Описание команды**
mulhs, mulhu	RT,RA,RB	В RT помещаются старшие 32 бита результата умножения RA и RB (знакового и беззнакового)
not	RT,RA	В RT помещается побитовое дополнение RA до единицы
shl, shr, shrs	RT, RA, RB	В RT помещается значение RA, сдвинутое влево или вправо; величина сдвига задается шестью младшими битами второго операнда (RB). При выполнении команды shrs освободившиеся биты заполняются содержимым знакового разряда, в остальных командах освободившиеся биты заполняются нулями. (Значение величины сдвига вычисляется по модулю 64)
shli,shri,shrsi	RT, RA, Iu	В RT помещается значение RA, сдвинутое влево или вправо на величину, задаваемую пятью младшими битами непосредственно заданного значения I
stb,sth,stw	RS, d(RA)	Сохранение байта, полуслова или слова из регистра RS в ячейку памяти по адресу RA + d, где d — непосредственно заданное 16-битовое знаковое значение

Везде в описаниях команд исходные операнды RA и RB представляют собой содержимое регистров.

В реальной машине обязательно должны быть команды ветвления и обращения к подпрограммам, команды перехода по адресу, содержащемуся в регистре (для возврата из подпрограмм и обработки оператора выбора альтернативы типа switch), а также, возможно, ряд команд для работы с регистрами специального назначения. Конечно же, должны быть привилегированные команды и команды вызова служб супервизора. Кроме того, вероятно наличие команд для работы с числами с плавающей точкой.

Краткое описание некоторых дополнительных команд, которые может иметь RISC-компьютер, приведено в табл. 1.3.

Таблица 1.3. Дополнительные команды полного набора RISC-команд

Мнемокод команды	Операнды	Описание команды
abs, nabs	RT, RA	RT получает абсолютное значение (или отрицательное абсолютное значение) RA
andc, eqv, nand, nor, orc	RT, RA, RB	Побитовое *и с дополнением, эквивалентность, отрицание и, отрицание или* и *или с дополнением*
extr	RT, RA, I, L	Извлечение битов от I до I+L-1 из RA и размещение их в младших битах RT; остальные разряды заполняются нулями
extrs	RT, RA, I, L	Аналогична extr, но свободные биты заполняются содержимым бита знака

Мнемокод команды	Операнды	Описание команды
ins	RT, RA, I, L	Вставляет биты регистра RA от 0 до L-1 в биты от I до I+L-1 приемника RT
nlz	RT, RA	RT содержит количество старших нулевых битов RA (от 0 до 32)
pop	RT, RA	RT содержит количество единичных битов RA (от 0 до 32)
ldb	RT, d(RA)	Загрузка знакового байта из ячейки памяти по адресу RA + d в RT, где d — непосредственно задаваемое 16-битовое знаковое значение
moveq, movne, movlt, movle, movgt, movge	RT, RA, RB	В RT помещается значение RB, если RA=0 (RA≠0 и т.д.). Если условие не выполняется, содержимое RT не изменяется
shlr, shrr	RT, RA, RB	В RT помещается значение RA после циклического сдвига влево или вправо; величина сдвига задается пятью младшими битами RB
shlri, shrri	RT,RA,Iu	В RT помещается значение RA после циклического сдвига влево или вправо; величина сдвига задается пятью младшими битами непосредственно заданного значения
trpeq, trpne, trplt, trple, trpgt, trpge, trpltu, trpleu, trpgtu, trpgeu	RA, RB	Прерывание *(trap)*, если RA=RB (RA≠RB и т.д.)
trpieq, trpine, trpilt, trpile, trpigt, trpige	RA, I	Аналогичны trpeq и остальным командам серии с тем отличием, что второй операнд представляет собой непосредственно заданное 16-битовое знаковое значение
trpiequ, trpineu, trpiltu, trpileu, trpigtu, trpigeu	RA, Iu	Аналогичны trpltu и другим командам серии с тем отличием, что второй операнд представляет собой непосредственно заданное 16-битовое беззнаковое значение

На практике в языке ассемблера удобно иметь ряд "расширенных мнемоник", похожих на макросы, которые обычно выполняются как одна команда. Некоторые из них представлены в табл. 1.4.

Таблица 1.4. Расширенные мнемоники

Расширенная мнемоника	Расширение	Описание
b target	beq RO, target	Безусловный переход
li RT, I	См. пояснение в тексте	Загрузка непосредственно заданного числа, $-2^{31} \leq I < 2^{32}$
mov RT, RA	ori RT, RA, 0	Пересылка регистра RA в RT
neg RT, RA	sub RT, RO, RA	Отрицание (побитовое дополнение до двух)
subi RT, RA, I	addi RT, RA, -I	Вычитание непосредственно заданного значения ($I \neq -2^{15}$)

Команда загрузки непосредственно заданного значения расширяется до одной или двух команд, в зависимости от непосредственного значения I. Например, если $0 \leq I < 2^{16}$, можно применить команду ori с использованием регистра RO. Если $-2^{15} \leq I < 0$, можно обойтись командой addi с регистром RO. Если младшие 16 битов I — нулевые, используем команду addis. Для остальных значений I выполняются две команды: например, после команды ori выполняется addis. (Иначе говоря, в последнем случае выполняется загрузка из памяти, но при оценке времени и места, которое требуется на выполнение этой макрокоманды, будем считать, что выполняются две элементарные арифметические команды.)

Существуют разные мнения о том, к какому набору RISC-команд следует отнести ту или иную команду — к базовому или к расширенному. Команды *беззнакового деления* и *получения остатка от деления* вполне можно было бы включить в расширенный набор RISC-команд. Команда *знакового сдвига вправо* — еще одна команда, редко используемая в тестах SPEC. Причина в том, что в С очень легко случайно использовать эти команды неправильно, например выполнив беззнаковое деление там, где используются знаковые операнды, или выполнить сдвиг вправо на величину со знаком (int), которая на самом деле является беззнаковой. Кстати, *не* следует выполнять деление целого знакового числа на степень 2 при помощи *знакового сдвига вправо;* так, вы должны добавить к результату 1, если делимое отрицательно и был выполнен сдвиг нескольких ненулевых битов.

Различия между основным и расширенным набором RISC-команд приводят ко множеству подобных спорных моментов, но не будем подробно останавливаться на этой теме.

Действие команд ограничивается двумя входными регистрами и одним выходным, что упрощает работу компьютера. Упрощается также работа оптимизирующего компилятора — ему не приходится обрабатывать инструкции с несколькими целевыми регистрами. Однако за такое упрощение приходится платить: если в программе требуется вычислить и частное, и остаток от деления двух чисел, то приходится выполнять две команды (деление и получение остатка). Стандартный алгоритм деления вместе с частным вычисляет и остаток от деления, поэтому на многих машинах и частное и остаток получаются в результате выполнения одной команды деления. Аналогичные замечания применимы и к случаю, когда при умножении двух слов получается двойное слово-произведение.

Создается впечатление, что команда *условной пересылки* (moveq) имеет только два входных операнда, хотя на самом деле их три. Поскольку результат выполнения этой команды зависит от содержимого регистров RT, RA и RB, при неупорядоченном выполнении команд RT следует интерпретировать и как *используемый,* и как *устанавливаемый* ре-

гистр одновременно. Представьте ситуацию, когда за командой, устанавливающей значение RT, следует команда *условной пересылки* и при этом нельзя потерять результат первой команды. Таким образом, разработчик машины может убрать команду *условной пересылки* из набора допустимых команд, тем самым исключив обработку команд с тремя (логическими) входными операндами. С другой стороны, команда *условной пересылки* снижает количество ветвлений в программе.

В данной книге формат команд не играет большой роли. Полное множество RISC-команд, описанное выше, вместе с командами для работы с числами с плавающей точкой и рядом команд супервизора может быть реализовано при помощи 32-битовых команд на компьютере с 32 регистрами общего назначения (5-битовые поля регистров). Если размер непосредственно заданных полей в командах сравнения, загрузки, сохранения и прерывания снизить до 14 бит, то это же множество команд может быть реализовано на компьютере с 64 регистрами общего назначения (с использованием 6-битовых полей регистров).

Время выполнения

Предполагается, что все команды выполняются за один такт процессора, за исключением команд умножения, деления и получения остатка от деления, для которых невозможно точно определить время выполнения. Команды перехода занимают один такт, независимо от того, был выполнен переход или нет.

Команда *загрузки непосредственного значения* выполняется за один или два такта, в зависимости от того, сколько элементарных арифметических команд требуется для формирования константы в регистре.

Команды *загрузки* и *сохранения* не часто упоминаются в данной книге, но будем считать, что они выполняются за один такт, пренебрегая при этом задержкой при загрузке (промежутком времени между моментом завершения команды в арифметическом модуле и моментом, когда данные становятся доступны следующей команде).

Однако зачастую одних только знаний о количестве тактов, используемых всеми арифметическими и логическими командами, недостаточно для правильной оценки времени выполнения программы. Выполнение программы часто замедляется вследствие задержек, возникающих при загрузке и выборке данных. Задержки такого рода не обсуждаются в книге, хотя и могут существенно влиять на скорость выполнения программ (причем это влияние постоянно возрастает). Другим источником сокращения времени выполнения является возможность распараллеливания вычислений на уровне команд, которая реализована практически на всех современных RISC-компьютерах, особенно на специализированных быстродействующих машинах.

В таких машинах имеется несколько исполнительных модулей и возможность диспетчеризации команд, что позволяет параллельное выполнение независимых команд (независимыми команды считаются тогда, когда ни одна из них не использует результаты других команд, команды не заносят значения в один и тот же регистр или бит состояния). Поскольку такие возможности компьютеров теперь уже не редкость, в книге много внимания уделяется независимым командам. Так, можно сказать, что некая формула может быть закодирована таким образом, что для вычисления потребуется выполнение восьми инструкций за пять тактов на машине с неограниченными возможностями распараллеливания вычислений на уровне команд. Это означает: если команды расположены в правильном порядке и машина имеет достаточное количество регистров и арифметических модулей, то она в принципе способна выполнить такую программу за пять тактов.

Более подробно обсуждать этот вопрос не имеет смысла, так как возможности машин в плане распараллеливания вычислений на уровне команд существенно различаются. Например, процессор ШМ RS/6000, созданный в 1992 году, имеет трехвходовой сумматор и может параллельно выполнять две следующие друг за другом команды сложения, даже если одна из них использует результаты другой команды (например, когда команда сложения использует результат команды сравнения или основной регистр команды загрузки). В качестве обратного примера можно рассмотреть простейший дешевый встраиваемый в различные системы компьютер, у которого регистр ввода-вывода имеет только один порт для чтения. Естественно, что такой машине для выполнения команды с двумя операндами-источниками потребуется дополнительный такт для повторного чтения регистра ввода-вывода. Однако если команда вьвдает операнд для команды, непосредственно следующей за ней, то этот операнд оказывается доступным и без чтения регистра ввода-вывода. На машинах такого типа возможно ускорение работы, если каждая команда предоставляет данные для следующей команды, т.е. если параллелизмы в коде отсутствуют.

ГЛАВА 2
ОСНОВЫ

2.1. Манипуляции с младшими битами

Многие из приведенных в этом разделе формул используются в следующих главах.

Для того чтобы обнулить крайний справа единичный бит (например, 01011000 ⇒ 01010000), используется формула

$$x \& (x-1) .$$

Эту формулу можно применять для проверки того, является ли беззнаковое целое степенью 2 (путем проверки результата вычислений на равенство 0).

Для проверки того, имеет ли беззнаковое целое число вид 2^n–1, можно воспользоваться следующей формулой:

$$x \& (x+1) .$$

Чтобы выделить в слове крайний справа единичный бит (например, 01011000 ⇒ 00001000, если такого бита нет, возвращает 0), используется формула

$$x \& (-x) .$$

Чтобы выделить в слове крайний справа нулевой бит (например, 10100111 ⇒ 00001000, если такого бита нет, возвращает 0), используется формула

$$\neg x \& (x+1) .$$

Для создания маски, идентифицирующей завершающие нулевые биты, можно воспользоваться одной из приведенных ниже формул. (Например, 01011000 ⇒ 000001 И, если исходное слово равно 0, все биты маски будут равны 1.)

$$\neg x \& (x-1) \text{,или}$$

$$\neg(x \mid -x) \text{,или}$$

$$(x \& -x)-1$$

Первая из этих формул позволяет распараллелить вычисления на уровне команд.

Приведенная далее формула используется для создания маски, идентифицирующей крайний справа единичный бит и все завершающие нулевые биты (например, 01011000 ⇒ 00001111, если исходное слово равно 0, все биты маски будут равны 1).

x®(x-1)

Очередная формула распространяет вправо крайний правый единичный бит (например, 01011000 ⇒ 01011111, если исходное слово равно 0, все биты результата будут равны 1).

$$x \mid (x-1)$$

Для обнуления крайней справа непрерывной подстроки единичных битов (например, 01011000⇒01000000) можно использовать формулу

$$((x \mid (x-1))+1) \& x .$$

Эта формула позволяет проверить, имеет ли положительное целое число вид 2^j-2^k, где $j > k > 0$ (в этом случае формула возвращает нулевой результат).

Для всех приведенных выше формул выполняется принцип дуализма: если в описаниях формул единицы заменить нулями (и нули единицами), а в самих формулах заменить x -1 на $x+1$ и обратно, $-x$ на $\neg(x+1)$, & на | и | на &, оставив без изменений только x и $\neg x$, то в результате получатся правильные выражения и описания. Например, после замен и преобразований в первой формуле получим дуальную ей формулу, которая будет выглядеть следующим образом.

Для того чтобы установить крайний справа нулевой бит (например, 10100111 ⇒ 10101111, если исходное слово равно 0, возвращает слово, состоящее из одних единиц), используется следующая формула:

$$x \mid (x+1).$$

Имеется простая проверка того, можно ли реализовать заданную функцию в виде последовательности команд *сложения, вычитания, и, или* и *отрицания* [59]. К перечисленным выше командам можно добавить другие команды из основного множества, которые, в свою очередь, являются комбинацией команд *сложения, вычитания, и, или* и *не,* например можно добавить команду *сдвиг влево* на определенную величину (которая эквивалентна последовательности команд *сложения)* или команду *умножения.* Команды, которые не могут быть выражены через команды *сложения, вычитания* и другие из основного списка, исключаются из рассмотрения. Критерий проверки следует из приведенной теоремы.

Теорема. *Функция, отображающая слова в слова, может быть реализована посредством операций побитового сложения, вычитания, и, или, отрицания тогда и только тогда, когда каждый бит результата зависит только от битов исходных операндов в той же позиции и правее нее.*

Иными словами, представьте ситуацию, когда вы вычисляете младший бит результирующего значения, исходя только из младших битов операндов. Затем вычисляется второй справа бит результата, для чего используются только два младших бита операндов, и т.д. Если таким образом можно получить результирующее значение, то данная функция может быть вычислена при помощи последовательности команд *сложения, и* и т.п. Если же таким образом (справа налево) вычислить функцию невозможно, то ее невозможно реализовать в виде последовательности указанных команд.

Наибольший интерес представляет утверждение, что любая из функций *сложения, вычитания, и, или* и *не* может быть вычислена справа налево, так что любая комбинация этих функций также обладает этим свойством, т.е. может быть вычислена справа налево.

Доказательство второй части теоремы достаточно громоздко, так что просто приведем конкретный пример. Предположим, что функция двух переменных x и y может быть вычислена справа налево, и пусть второй бит результата r вычисляется следующим образом:

$$r_2 = x_2 \mid (x_0 \,\&\, y_1). \tag{1}$$

(Биты пронумерованы справа налево от 0 до 31.) Так как второй бит результата является функцией только крайних справа битов исходных операндов (бита номер 2 и битов младше его), он также вычисляется справа налево.

Запишем машинные слова x, x, сдвинутое на два бита влево, и y, сдвинутое на один бит влево, как показано ниже. Добавим к записи маску, которая выделяет второй бит.

x_{31}	x_{30}	...	x_3	x_2	x_1	x_0
x_{29}	x_{28}	...	x_1	x_0	0	0
y_{30}	y_{29}	...	y_2	y_1	y_0	0
0	0	...	0	1	0	0
0	0	...	0	r_2	0	0

К строкам 2 и 3 применим операцию *побитового и,* затем, согласно формуле (1), к получившемуся слову применим операцию *побитового или со* строкой 1, после чего к полученному результату и маске (строка 4) применим операцию *побитового и.* В полученном таким образом слове все биты, кроме второго, будут нулевыми. Выполним подобные вычисления для получения остальных битов результата и объединим все 32 слова с помощью операции *или.* В итоге получим искомую функцию.

Такое построение не дает эффективной программы вычисления значений функции, всего лишь показывая, что искомая функция может быть выражена с использованием команд из базового списка.

Используя эту теорему, можно сразу же увидеть, что не существует такой последовательности команд, которая обнуляла бы в слове крайний слева единичный бит, так как для этого необходимо просмотреть биты, находящиеся слева от него (чтобы точно знать, что это действительно крайний слева единичный бит). Аналогично, не существует такой последовательности операций, которая бы давала сдвиг вправо, циклический сдвиг или сдвиг влево на переменную величину, а также могла подсчитать количество завершающих нулевых битов в слове (при подсчете завершающих нулевых битов младший бит результата должен быть равен 1, если это количество нечетно, и 0, если четно; так что необходимо просмотреть биты слева от младшего, чтобы выяснить его значение).

Одним из применений рассмотренной сортировки битов является задача поиска следующего числа, которое больше заданного, но имеет такое же количество единичных битов. Зачем это может понадобиться? Эта задача возникает при использовании битовых строк для представления подмножеств. Возможные члены множества перечислены в одномерном массиве, а подмножество представляет собой слово или последовательность слов, в которых бит i установлен равным 1, если i-й элемент принадлежит этому подмножеству. Тогда объединение множеств вычисляется с помощью *побитовой операции или* над битовыми строками, пересечение — с помощью операции *и* и т.д.

Иногда требуется выполнить определенные действия над всеми подмножествами заданной длины, что легко сделать с помощью функции, отображающей представленное числом заданное подмножество в следующее большее число, содержащее такое же количество единичных битов, что и исходное (строка подмножества интерпретируется как целое число).

Компактный алгоритм для этой задачи составлен Р.У. Госпером (R.W. Gosper) [25, item 175][1]. Пусть имеется некоторое слово x, представляющее собой подмножество. Идея в том, чтобы сначала найти в x крайнюю справа группу смежных единичных битов, следующую за нулевыми битами, а затем увеличить представленную этой группой величину таким образом, чтобы количество единичных битов осталось неизменным. Например, строка xxx011110000, где xxx — произвольные биты, преобразуется в строку xxx100000111. Алгоритм работает следующим образом. Сначала в x определяется самый младший единичный бит, т.е. вычисляется $s = x \& -x$ (в результате получаем 000000010000). Затем эта строка складывается с x, давая r=xxx100000000, в которой

[1] Вариант этого алгоритма имеется в [32], разд.7.6.7.

получен первый единичный бит результата. Чтобы получить остальные биты, установим $n-1$ младших битов слова равными 1, где я — размер крайней правой группы единичных битов в исходном слове x. Для этого над r и x выполняется операция *исключающего или*, что в нашем примере дает строку 0001111110000.

В полученной строке содержится больше единичных битов, чем в исходной, поэтому выполним еще одно преобразование. Поделим искомое число на s, что эквивалентно его сдвигу вправо (s — степень 2), сдвинем его вправо еще на два разряда, что отбросит еще два нежелательных бита, и выполним операцию побитового *или* над полученным числом и r.

В записи компьютерной алгебры результат y получается следующим образом.

$$\begin{aligned} s &\leftarrow x \,\&\, -x \\ r &\leftarrow s + x \\ y &\leftarrow r \mid \left(\left((x \oplus r) \overset{u}{\gg} 2\right) \overset{u}{\div} s\right) \end{aligned} \tag{2}$$

В листинге 2.1 представлена законченная процедура на языке C, выполняющая семь базовых RISC-команд, одной из которых является команда деления. (Данная процедура неприменима для $x=0$, так как в этом случае получается деление на 0.)

Листинг 2.1. Вычисление следующего большего числа с тем же количеством единичных битов

```
unsigned snoob(unsigned x) {
    unsigned smallest, ripple, ones;
                                     // x = xxx0 1111 0000
    smallest = x & -x;               //     0000 0001 0000
    ripple   = x + smallest;         //     xxx1 0000 0000
    ones     = x ^ripple;            //     0001 1111 0000
    ones     = (ones >> 2)/smallest; //     0000 0000 0111
    return   ripple | ones;          //     xxx1 0000 0111
}
```

Если деление выполняется медленно, но у вас есть быстрый способ вычисления *количества завершающих нулевых битов* $\mathrm{ntz}(x)$, *количества ведущих нулевых битов* $\mathrm{nlz}(x)$ или функции *степени заполнения* $\mathrm{pop}(x)$ (количество единичных битов в x), то последнюю строку в (2) можно заменить одной из приведенных ниже (однако первые два метода дают сбой на последнем шаге для одноэлементного подмножества на машинах со сдвигом по модулю 32).

$$\begin{aligned} y &\leftarrow r \mid \left((x \oplus r) \overset{u}{\gg} (2 + \mathrm{ntz}(x))\right) \\ y &\leftarrow r \mid \left((x \oplus r) \overset{u}{\gg} (33 - \mathrm{nlz}(s))\right) \\ y &\leftarrow r \mid \left((1 \ll (\mathrm{pop}(x \oplus r) - 2)) - 1\right) \end{aligned}$$

2.2. Сложение и логические операции

Предполагается, что читатель знаком с элементарными тождествами обычной и булевой алгебры. Ниже приводится набор аналогичных тождеств для операций сложения и вычитания в комбинации с логическими операциями.

а) $-x = \neg x + 1$

б) $ = \neg(x - 1)$

в) $\neg x = -x - 1$

г) $-\neg x = x+1$

д) $\neg -x = x-1$

е) $x+y = x-\neg y-1$

ж) $= (x \oplus y)+2(x \& y)$

з) $= (x \setminus y)+(x \& y)$

и) $= 2(x \mid y)-(x \oplus y)$

к) $x-y = x+\neg y+1$

л) $= (x \oplus y)-2(\neg x \& y)$

м) $= (x \& \neg y)-(\neg x \& y)$

н) $= 2(x \& \neg y)-(x \oplus y)$

о) $x \oplus y = (x \setminus y)-(x \& y)$

п) $x \& \neg y = (x \mid y)-y$

Р) $= x-(x \& y)$

с) $\neg(x-y) = y-x-1$

т) $= \neg x+y$

У) $x \equiv y = (x \& y)-(x \mid y)-1$

Ф) $= (x \& y)+\neg(x \mid y)$

Х) $x \mid y = (x \& \neg y)+y$

ц) $x \& y = (\neg x \mid y)-\neg x$

Равенство (г) можно повторно применять к самому себе, например: $-\neg-\neg x = x+2$ и т.д. Аналогично, после повторного использования равенства (д) получаем: $\neg-\neg- x = x-2$. Таким образом, чтобы добавить к x или вычесть из него некоторую константу, можно воспользоваться только этими выражениями дополнения.

Равенство (е) дуально равенству (к), которое представляет собой известное отношение, позволяющее получить вычитающее устройство из сумматора.

Равенства (ж) и (з) взяты из [25, item 23]. В равенстве (ж) сумма x и y вычисляется следующим образом: сначала x и y суммируются без учета переносов $(x \Phi y)$, а затем к полученному результату добавляются переносы. Равенство (з) изменяет операнды перед сложением так, что в любом разряде все комбинации типа $0+1$ заменяются на 1 +0.

Можно показать, что при обычном сложении двоичных чисел с независимыми битами вероятность возникновения переноса в любом разряде равна 0.5. Однако, если при создании сумматора используется предварительная подготовка входных данных, как при сложении по формуле (ж), вероятность появления переноса становится равной 0.25. При разработке сумматора вероятность переноса значения не имеет; самой важной характеристикой при этом является максимальное количество логических схем, через которые может потребоваться пройти переносу, а использование равенства (ж) позволяет свести это количество всего лишь к одной схеме.

Равенства (л) и (м) дуальны равенствам (ж) и (з) для вычитания. В (л) сначала выполняется вычитание без заемов $(x \oplus y)$, а затем из полученного результата вычитаются заемы. Аналогично, равенство (м) изменяет операнды перед вычитанием таким образом, что все комбинации типа $1-1$ заменяются на 0-0.

Равенство (о) показывает, как реализовать *исключающее или,* используя всего три базовых RISC-команды. Использование логики *и–или–не* потребует выполнения четырех команд $((x \mid y) \,\&\, \neg(x \,\&\, y))$. Аналогично, равенства (х) и (ц) реализуют операции *и* и *или* с помощью трех элементарных команд (хотя по законам де Моргана (DeMorgan) их требуется четыре.

2.3. Неравенства с логическими и арифметическими выражениями

Неравенства с двоичными логическими выражениями, значения которых интерпретируются как целые беззнаковые величины, выводятся практически тривиально. Вот пара примеров:

$$(x \oplus y) \overset{u}{\le} (x \mid y) \text{ и}$$

$$(x \,\&\, y) \overset{u}{\le} (x \equiv y).$$

Их легко получить из табл. 2.1, в которой представлены все логические операции.

Таблица 2.1. Результаты работы 16 логических операций

x	y	о	$x\&y$	$x\&\neg y$	ц	$\neg x\&x$	y	$x\oplus y$	$x\mid y$	$\neg(x\mid y)$	$x\equiv y$	$\neg y$	$x\mid\neg y$	$\neg x$	$\neg x\mid y$	Т $\neg(x\&y)$	–
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	0	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Пусть функции $f(x,y)$ и $g(x,y)$ представлены двумя столбцами табл. 2.1. Если в каждой строке, где $f(x,y)$ равна 1, $g(x,y)$ также равна 1, то для любых значений (x,y) выполняется соотношение $f(x,y) \overset{u}{\le} g(x,y)$. Очевидно, что это неравенство справедливо и для побитовых логических операций. Легко вывести и более сложные соотношения (многие из которых тривиальны), например: $(x \,\&\, y) \overset{u}{\le} x \overset{u}{\le} (x \mid \neg y)$ и т.п. Если же значения функций $f(x,y)$ и $g(x,y)$ в одной строке таблицы равны 0 и 1, а в другой — 1 и 0 соответственно, то между этими логическими функциями не существует отношения неравенства. Таким образом, всегда можно выяснить, выполняется ли для каких-то функций f и g соотношение $f(x,y) \overset{u}{\le} g(x,y)$ или нет.

При работе с неравенствами требуется внимание. Например, в обычной арифметике, если $x+y \le a$ и $z \le x$, *то* $z+y \le a$. Но этот вывод становится неверным, если "+" заменить оператором *или.*

Неравенства, включающие как логические, так и арифметические выражения, более интересны. Ниже приведены некоторые из них.

а) $(x \mid y) \overset{u}{\geq} \max(x, y)$

б) $(x \,\&\, y) \overset{u}{\leq} \min(x, y)$

в) $(x \mid y) \overset{u}{\leq} x + y$ Если сложение не вызывает переполнения

г) $(x \mid y) \overset{u}{>} x + y$ Если сложение вызывает переполнение

д) $|x - y| \overset{u}{\leq} (x \oplus y)$

Доказать все эти неравенства достаточно просто, за исключением, возможно, только неравенства (д). Под $|x - y|$ подразумевается абсолютное значение $x - y$, которое может быть вычислено в области беззнаковых чисел как $\max(x, y) - \min(x, y)$. Это неравенство можно доказать по индукции по длинам x и y (доказать неравенство будет проще, если длины x и y расширять справа налево, а не наоборот).

2.4. Абсолютное значение

Если нет отдельной команды вычисления абсолютного значения, ее можно реализовать тремя или четырьмя командами (без ветвления). Сначала вычисляется значение $y \leftarrow x \overset{s}{\gg} 31$, а затем одно из перечисленных ниже выражений (разумеется, $2x$ означает $x + x$, или $x \ll 1$).

abs	nabs
$(x \oplus y) - y$	$y - (x \oplus y)$
$(x + y) \oplus y$	$(y - x) \oplus y$
$x - (2x \& y)$	$(2x \& y) - x$

Если у вас есть возможность быстрого умножения на переменную, значение которой равно ± 1, можно поступить следующим образом:

$$\left(\left(x \overset{s}{\gg} 30\right) \mid 1\right) * x.$$

2.5. Распространение знака

Под "распространением знака" *(sign extension)* подразумевается наличие бита в определенной позиции слова, выступающего в роли бита знака, и распространение этого бита влево при игнорировании всех остальных битов слова. Стандартный способ решения этой задачи состоит в *логическом сдвиге влево,* за которым следует *знаковый сдвиг вправо.* Однако, если эти команды работают медленно или их вообще нет на вашем компьютере, можно поступить иначе. Ниже приведены примеры распространения влево бита в седьмой позиции.

$$((x + \text{0x00000080}) \,\&\, \text{0x000000FF}) - \text{0x00000080}$$

$$((x \,\&\, \text{0x000000FF}) \oplus \text{0x00000080}) - \text{0x00000080}$$

Вместо “+” можно использовать “–” или “⊕”. Если известно, что все ненужные старшие биты нулевые, вторая формула оказывается более практичной, так как в этом случае можно не выполнять команду *и*.

2.6. Знаковый сдвиг вправо на основе беззнакового сдвига

Если на машине нет команды *знакового сдвига вправо (shift right signed)*, ее можно заменить другими командами. Первая формула взята из [21], вторая основана на той же идее, что и первая. На 64-разрядной машине первые четыре формулы применимы для $0 \le n \le 31$, а последняя — для $0 \le n \le 63$. В принципе она применима для любых n, где под "применима" подразумевается "рассматривает величину сдвига по тому же модулю, что и логический сдвиг".

Для переменной n реализация каждой формулы требует пяти или шести команд из базового множества RISC-команд.

$$\left((x + \mathbf{0x80000000}) \overset{u}{\gg} n\right) - \left(\mathbf{0x80000000} \overset{u}{\gg} n\right)$$

$$t \leftarrow \mathbf{0x80000000} \overset{u}{\gg} n; \quad \left(\left(x \overset{u}{\gg} n\right) \oplus t\right) - t$$

$$t \leftarrow (x \,\&\, \mathbf{0x80000000}) \overset{u}{\gg} n; \quad \left(x \overset{u}{\gg} n\right) - (t + t)$$

$$\left(x \overset{u}{\gg} n\right) \mid \left(-\left(x \overset{u}{\gg} 31\right) \ll 31 - n\right)$$

$$t \leftarrow -\left(x \overset{u}{\gg} 31\right); \quad \left(\left(x \oplus t\right) \overset{u}{\gg} n\right) \oplus t$$

В первых двух формулах выражение $\mathbf{0x80000000} \overset{u}{\gg} n$ можно заменить на $1 \ll 31 - n$.

Если n — константа, то для реализации первых двух формул на большинстве машин требуется всего лишь три команды. Если $n = 31$, функция может быть вычислена с помощью двух команд: $-\left(x \overset{u}{\gg} 31\right)$.

2.7. Функция *sign*

Функция sign, или *signum*, определяется как

$$\operatorname{sign}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$$

На большинстве машин ее можно вычислить с помощью четырех команд [29].

$$\left(x \overset{s}{\gg} 31\right) \mid \left(-x \overset{u}{\gg} 31\right)$$

Если на машине нет команды *знакового сдвига вправо,* вместо нее можно использовать последнее выражение из раздела 2.6. В результате получается элегантная симметричная формула (вычисляемая за пять команд).

$$-\left(x \overset{u}{\gg} 31\right) \mid \left(-x \overset{u}{\gg} 31\right)$$

Наличие команд сравнения допускает решения из трех инструкций.

$$\begin{matrix}(x > 0) - (x < 0) \\ (x \ge 0) - (x \le 0)\end{matrix} \qquad (3)$$

И последнее замечание: почти всегда работает формула $\left(-x \overset{u}{\gg} 31\right) - \left(x \overset{u}{\gg} 31\right)$; ошибка возникает только при $x = -2^{31}$.

2.8. Трехзначная функция сравнения

Трехзначная функция сравнения представляет собой определенное обобщение функции *sign* и определяется следующим образом:

$$\text{cmp}(x, y) = \begin{cases} -1, & x < y, \\ 0, & x = y, \\ 1, & x > y. \end{cases}$$

Данное определение справедливо для любых переменных — и знаковых и беззнаковых. Все, что говорится в этом разделе, если явно не оговорено иное, применимо к обоим случаям.

Использование команд сравнения позволяет реализовать эту функцию за три команды, как очевидное обобщение выражений (3).

$$\begin{matrix}(x > y) - (x < y) \\ (x \ge y) - (x \le y)\end{matrix}$$

Решение для беззнаковых целых чисел на компьютере PowerPC приведено ниже [10]. На этой машине "перенос" означает "не заем".

```
subf  R5, Ry, Rx # R5 <-- Rx-Ry
subfc R6, Rx, Ry # R6 <-- Ry- Rx, установлен перенос
subfe R7, Ry, Rx # R7 <-- Rx-Ry+перенос, установлен перенос
subfe R8, R7, R5 # R8 <-- R5-R7+перенос, (установлен перенос)
```

Если ограничиться командами из базового набора RISC-команд, то эффективного метода вычисления этой функции нет. Для вычисления $x < y$, $x \le y$ и других команд сравнения потребуется выполнить пять команд (см. раздел 2.11), что приводит к решению, содержащему 12 команд (при использовании определенной общности вычислений $x < y$ и $y < x$). На RISC-машинах с базовым набором команд лучшим путем решения будет использование команд сравнения и ветвления (в худшем случае потребуется выполнить шесть команд).

2.9. Перенос знака

Функция *переноса знака* (известная в Fortran как ISIGN), определяется следующим образом:

$$\text{ISIGN}(x, y) = \begin{cases} \text{abs}(x), & y \ge 0, \\ -\text{abs}(x), & y < 0. \end{cases}$$

На большинстве машин эта функция может быть вычислена (по модулю 32) за четыре команды.

$$t \leftarrow y \overset{s}{\gg} 31; \qquad\qquad t \leftarrow (x \oplus y) \overset{s}{\gg} 31;$$
$$\text{ISIGN}(x, y) = (\text{abs}(x) \oplus t) - t = \qquad \text{ISIGN}(x, y) = (x \oplus t) - t =$$
$$= (\text{abs}(x) + t) \oplus t \qquad\qquad = (x + t) \oplus t$$

2.10. Декодирование поля "0 означает 2**n"

Иногда 0 или отрицательное значение для некоторой величины не имеет смысла. Поэтому она кодируется нулевым значением в n-м поле, которое интерпретируется как 2^n, а ненулевое значение имеет обычный двоичный смысл. Например, у PowerPC длина поля в команде загрузки непосредственно заданной строки слов (`lswi`) занимает 5 битов. Нелогично использовать эту команду для загрузки нуля байтов, поскольку длина представляет собой непосредственно заданную величину, но было бы очень полезно иметь возможность загрузить 32 байта. Длина поля кодируется значениями от 0 до 31, которые могут, например, означать длины от 1 до 32, но соглашение "нуль означает 32" приводит к более простой логике, в особенности когда процессор должен также поддерживать соответствующую команду с переменной (задаваемой в регистре) длиной, которая выполняет простое бинарное кодирование (например, команду `lswx` на PowerPC).

Кодирование целых чисел от 1 до 2^n в поле, ноль в котором означает 2^n, тривиально: просто маскировать биты целого числа с помощью маски 2^n-1. Выполнить же декодирование без проверок и переходов не так-то просто. Ниже приведено несколько возможных вариантов для работы с третьим битом. Все эти варианты требуют выполнения трех команд, не считая возможных загрузок констант.

$$((x-1)\,\&\,7)+1 \qquad ((x-1)\,|\,{-8})+9 \qquad ((x+7)\,|\,8)-7 \qquad 8-(-x\,\&\,7)$$
$$((x+7)\,\&\,7)+1 \qquad ((x+7)\,|\,{-8})+9 \qquad ((x-1)\,\&\,8)+x \qquad -(-x\,|\,8)$$

2.11. Предикаты сравнения

Предикаты сравнения представляют собой функции, которые сравнивают две величины и возвращают однобитовый результат, равный 1, если проверяемое отношение истинно, и 0, если ложно. В этом разделе приводится несколько методов вычисления результата сравнения с размещением его в бите знака (эти методы не используют команд ветвления). Чтобы получить значение 1 или 0, используемое в некоторых языках (например, в С), после вычисления следует использовать команду *сдвига вправо* на 31 бит. Для получения результата -1/0, используемого в некоторых других языках (например, Basic), выполняется команда *знакового сдвига вправо* на 31 бит.

На таких машинах, как MIPS, Compaq Alpha, и нашей модели RISC есть команды сравнения, которые непосредственно вычисляют большинство отношений и помещают однобитовый результат 0/1 в регистр общего назначения, а потому для таких машин приводимые ниже формулы не представляют особого интереса.

При вычислении функций сравнения двух операндов весьма полезной оказывается машинная команда "nabs", вычисляющая отрицательное абсолютное значение. В отличие от функции абсолютного значения, она никогда не приводит к переполнению. Если среди ко-

манд машины нет функции "nabs", но есть более привычная команда **"abs"**, то вместо $\text{nabs}(x)$ можно использовать $-\text{abs}(x)$. Если x — максимальное отрицательное число, переполнение возникает дважды, но итоговый результат получается правильный (предполагается, что абсолютное значение и отрицание максимального отрицательного числа дают одинаковый результат). Поскольку на многих машинах нет ни команды "abs", ни команды "nabs", приведем и альтернативные формулы, в которых эти функции не применяются.

Используемая ниже функция "nlz" возвращает количество ведущих нулевых битов аргумента. Описание функции doz *(разность или ноль)* приводится в разделе 2.18.

$$x = y: \quad \text{abs}(x - y) - 1$$

$$\text{abs}(x - y + \text{0x80000000})$$

$$\text{nlz}(x - y) \ll 26$$

$$-\left(\text{nlz}(x - y) \overset{u}{\gg} 5\right)$$

$$\neg(x - y \mid y - x)$$

$$x \neq y: \quad \text{nabs}(x - y)$$

$$\text{nlz}(x - y) - 32$$

$$x - y \backslash y - x$$

$$x < y: \quad (x - y) \oplus \left[(x \oplus y) \,\&\, ((x - y) \oplus x)\right]$$

$$(x \,\&\, \neg y) \mid ((x \equiv y) \,\&\, (x - y))$$

$$\text{nabs}(\text{doz}(y, x)) \qquad [24]$$

$$x \leq y: \quad (x \backslash \neg y) \,\&\, ((x \oplus y) \mid \neg(y - x))$$

$$\left((x \equiv y) \overset{s}{\gg} 1\right) + \left(x \,\&\, \neg y\right) \qquad [24]$$

$$x < y: \quad (\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x - y))$$

$$(\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y))$$

$$x \overset{u}{\leq} y: \quad (\neg x \mid y) \,\&\, ((x \oplus y) \mid \neg(y - x))$$

Для $x > y$, $x \geq y$ и прочих отношений необходимо в соответствующих формулах поменять местами x и y. Команду *сложения с* 0x80000000 можно заменить любой другой командой, которая инвертирует значение старшего бита.

Другой класс формул основан на том, что предикат $x < y$ вычисляется по знаку величины $x/2 - y/2$, а вычисление данной разности не может привести к переполнению. В тех случаях, когда сдвиги приводят к потере информации, результат можно уточнить путем вычитания единицы.

$$x < y: \quad \left(x \overset{s}{\gg} 1\right) - \left(y \overset{s}{\gg} 1\right) - \left(\neg x \,\&\, y \,\&\, 1\right)$$

$$x \overset{u}{<} y: \quad \left(x \overset{u}{\gg} 1\right) - \left(y \overset{u}{\gg} 1\right) - \left(\neg x \,\&\, y \,\&\, 1\right)$$

Большинство машин вычисляют эти отношения при помощи семи команд (или шести при наличии команды *и-не),* что ничем не лучше **результата,** достигнутого в предыдущих формулах (от пяти до семи команд в зависимости от полноты используемого множества команд).

Формулы, в которых используется функция "**nlz**", взяты из [56]. Ее использование позволяет вычислить результат отношения $x = y$ *в* виде 1 или 0 всего за три команды.

$$\text{nlz}(x-y) \overset{u}{\gg} 5$$

Команды знакового сравнения с 0 встречаются достаточно часто, что заслуживает отдельного рассмотрения. Ниже приводится несколько формул, которые в основном непосредственно выведены из предыдущих выражений. Так же, как и ранее, результат сравнения отображается в позиции бита знака.

$$\begin{array}{ll}
x = 0: & \text{abs}(x) - 1 \\
 & \text{abs}(x + \mathbf{0x80000000}) \\
 & \text{nlz}(x) \ll 26 \\
 & -\left(\text{nlz}(x) \overset{u}{\gg} 5\right) \\
 & \neg(x \mid -x) \\
 & \neg x \,\&\, (x-1) \\
x \neq 0: & \text{nabs}(x) \\
 & \text{nlz}(x) - 32 \\
 & x \mid -x \\
 & \left(x \overset{u}{\gg} 1\right) - * \qquad\qquad [10] \\
x < 0: & x \\
x \leq 0: & x \backslash (x-1) \\
 & x \mid \neg -x \\
x > 0: & x \oplus \text{nabs}(x) \\
 & \left(x \overset{s}{\gg} 1\right) - x \\
 & -x \,\&\, \neg x \\
x \geq 0: & \neg x
\end{array}$$

Знаковое сравнение можно получить из беззнакового, увеличивая сравниваемые величины на 2^{31}. Обратное преобразование также справедливо. Таким образом получаем

$$x < y = x + 2^{31} \overset{u}{<} y + 2^{31},$$
$$x \overset{u}{<} y = x - 2^{31} < y - 2^{31}.$$

Аналогичные соотношения выполняются для $\leq$, $\overset{u}{<}$ и т.д. В данном случае прибавление и вычитание числа 2^{1} — эквивалентные действия, так как обе операции инвертируют значение знакового разряда.

Еще один путь получения знакового сравнения из беззнакового основан на том, что если x и y имеют одинаковые знаки, то $x < y = x \overset{u}{<} y$, а если их знаки различны, то $x<y = x \overset{u}{>} y$ [42]. В этом случае также справедливы обратные преобразования; таким образом, получаем

$$x < y = \left(x \overset{u}{<} y \right) \oplus x_{31} \oplus y_{31},$$

$$x \overset{u}{<} y = (x < y) \oplus x_{31} \oplus y_{31},$$

где x_{31} и y_{31} представляют собой знаковые биты x и y соответственно. Аналогичные выражения получаются для отношений $\leq$, $\overset{u}{<}$ и других.

Применение описанных способов позволяет вычислить все стандартные команды сравнения (кроме $=$ и $\neq$) через другие команды сравнения с использованием не более трех дополнительных команд на большинстве машин. Например, возьмем в качестве исходной команды сравнения отношение $x \overset{u}{\leq} y$, поскольку реализовать его проще других команд (как бит переноса $y - x$). Тогда остальные команды сравнения можно реализовать следующим образом:

$$x<y = \neg\left(y + 2^{31} \overset{u}{\leq} x + 2^{31} \right)$$

$$x \leq y = x + 2^{31} \overset{u}{\leq} y + 2^{31}$$

$$x > y = \neg\left(x + 2^{31} \overset{u}{\leq} y + 2^{31} \right)$$

$$x \geq y = y + 2^{31} \overset{u}{\leq} x + 2^{31}$$

$$x \overset{u}{<} y = \neg\left(y \overset{u}{\leq} x \right)$$

$$x \overset{u}{>} y = \neg\left(x \overset{u}{\leq} y \right)$$

$$x \overset{u}{\geq} y = y \overset{u}{\leq} x$$

Команды сравнения и бит переноса

Если процессор может легко поместить бит переноса в регистр общего назначения, то для некоторых операций отношения это позволяет получить очень лаконичный код. Ниже приводятся выражения такого рода для некоторых отношений. Запись carry*(выражение)* означает, что разряд переноса генерируется при выполнении команды *выражение*. Предполагается, что бит переноса для разности $x - y$ тот же, что и на выходе сумматора для выражения $x + \bar{y} + 1$, и является дополнением "заема".

$x = y$: $\text{carry}\left(0 - (x - y)\right)$, или

$\text{carry}\left((x + \bar{y}) + 1\right)$, или

$\text{carry}\left((x - y - 1) + 1\right)$

$x \neq y$: $\text{carry}\left((x - y) - 1\right) = \text{carry}\left((x - y) + (-1)\right)$

$x < y$: $\neg\text{carry}\left((x + 2^{31}) - (y + 2^{31})\right)$

$x \le y$: $\quad \text{carry}\left(\left(y+2^{31}\right)-\left(x+2^{31}\right)\right)$

$x < y$: $\quad \neg\text{carry}(x-y)$

$x \le y$: $\quad \text{carry}(y-x)$

$x = 0$: $\quad \text{carry}(0-x)$, или $\text{carry}\left(\bar{x}+1\right)$

$x \ne 0$: $\quad \text{carry}(x-1) = \text{carry}(x+(-1))$

$x < 0$: $\quad \text{carry}(x+x)$

$x \le 0$: $\quad \text{carry}\left(2^{31}-\left(x+2^{31}\right)\right)$

Для $x > y$ используется дополнение выражения для $x \le y$; то же справедливо и для других типов отношения "больше".

При вычислении условных выражений на IBM RS/6000 и близком к нему PowerPC была использована программа GNU Superoptimizer [17]. У RS/6000 имеются команды $\text{abs}(x)$, $\text{nabs}(x)$, $\text{doz}(x,y)$ и ряд различных команд сложения и вычитания с использованием переноса. Как выяснилось, RS/6000 может вычислить все целочисленные условные выражения с помощью не более чем трех элементарных (выполняющихся за один такт) команд — результат, удививший даже самих создателей машины. В состав "всех условных выражений" входят шесть команд знакового сравнения и четыре команды беззнакового сравнения, а также их версии, в которых второй операнд равен нулю (при этом каждая из команд имеет два варианта — возвращающий результат 1/0 и возвращающий результат -1/0). Power PC, где нет функций $\text{abs}(x)$, $\text{nabs}(x)$ и $\text{doz}(x,y)$, вычисляет все перечисленные условные выражения не более чем за четыре элементарные команды.

Вычисление отношений

Большинство компьютеров возвращают однобитовый результат операции сравнения, который обычно размещается в "регистре условий", а на некоторых машинах (как, например, в нашей RISC-модели) — в регистре общего назначения. В любом случае вычисление отношений зачастую реализуется вычитанием операндов и некоторыми действиями над битами полученного результата, чтобы получить окончательный однобитовый результат работы команды сравнения.

Рассмотрим логику описанных выше действий. Пусть вместо разности $x - y$ компьютер вычисляет сумму $x + \bar{y} + 1$, и в результате этих вычислений получаются следующие величины:

Co - перенос из старшего разряда в разряд переноса;
C_i - перенос в старший разряд;
N - бит знака результата;
Z - величина, равная 1, если полученный результат (за исключением Co) нуле вой, и 0 в противоположном случае.

Таким образом, в системе обозначений булевой алгебры для различных отношений получим приведенные ниже выражения (расположение рядом двух величин подразумевает операцию *и*, а оператор "+" означает операцию *или).*

$V{:}\quad C_i \oplus C_0$ (знаковое переполнение)

$x = y:\quad Z$

$x \neq y:\quad \overline{Z}$

$x < y:\quad N \circledR V$

$x \leq y:\quad (N \oplus V) + Z$

$x > y:\quad (N \equiv V)\overline{Z}$

$x \geq y:\quad N \equiv V$

$x \overset{u}{<} y:\quad \overline{C_0}$

$x \overset{u}{\leq} y:\quad \overline{C_0} + Z$

$x \overset{u}{>} y:\quad C_0\overline{Z}$

$x \overset{u}{\geq} y:\quad C_0$

2.12. Обнаружение переполнения

"Переполнение" означает, что результат арифметической операции либо слишком велик, либо слишком мал, чтобы его можно было корректно представить в выходном регистре. В этом разделе обсуждаются методы, которые позволяют определить, когда возникнет переполнение, не используя при этом биты состояния, предназначенные для этой цели. Эти методы имеют особое значение на тех машинах (например, MIPS), где биты состояния отсутствуют (даже при наличии таких битов обращение к ним из языков высокого уровня, как правило, затруднено или даже невозможно).

Знаковое сложение и вычитание

Если в результате сложения и вычитания целых чисел произошло переполнение, то, как правило, старший бит результата отбрасывается и сумматор выдает только младшие биты. Переполнение при сложении целых знаковых чисел возникает тогда и только тогда, когда операнды имеют одинаковые знаки, а знак суммы противоположен знаку операндов. Удивительно, но это правило справедливо даже тогда, когда в сумматоре был выполнен перенос, т.е. при вычислении суммы $x + y + 1$. Данное правило играет важную роль при сложении знаковых целых чисел, состоящих из нескольких слов, так как в этом случае последнее сложение представляет собой знаковое сложение двух полных слов и бита переноса, значение которого может быть равно 0 или $+1$.

Для доказательства этого правила предположим, что слагаемые x и y представляют собой целые знаковые значения, состоящие из одного слова, а бит переноса c равен 0 или 1. Предположим также для простоты, что сложение выполняется на 4-разрядной машине. Если x и y имеют разные знаки, то $-8 \leq x \leq -1$ и $0 \leq y \leq 7$ (аналогичные границы получаются и при отрицательном y и неотрицательном x). В любом случае после сложения этих неравенств и необязательного добавления c получим $-8 \leq x + y + c \leq 7$.

Как видите, сумма может быть представлена как 4-разрядное целое число со знаком; следовательно, при сложении операндов с разными знаками переполнения не будет.

Теперь предположим, что x и y имеют одинаковые знаки. Здесь возможны два случая.

(а)	(б)
$-8 < x < -1$	$0 < x < 7$

$$-8 \le y \le -1 \qquad\qquad 0 \le y \le 7$$

Таким образом:

$$\text{(а)}\quad -16 \le x+y+c \le -1 \qquad\qquad \text{(б)}\quad 0 \le x+y+c \le 15$$

Переполнение возникает тогда, когда сумма не может быть представлена в виде 4-битового целого числа со знаком, т.е. если

$$\text{(а)}\quad -16 \le x+y+c \le -9 \qquad\qquad \text{(б)}\quad 8 \le x+y+c \le 15$$

В случае (а) старший разряд 4-битовой суммы равен 0, т.е. противоположен знакам x и y. В случае (б) старший бит 4-битовой суммы равен 1, т.е. также противоположен знакам x и y.

При вычитании целых чисел, состоящих из нескольких слов, нас интересует разность $x-y-c$, где c равно 0 или 1, причем 1 означает, что был выполнен заем. Проводя анализ, аналогичный только что рассмотренному, можно увидеть, что при вычислении выражения $x-y-c$ переполнение будет тогда и только тогда, когда x и y имеют разные знаки и знак разности противоположен знаку x (или, что то же самое, имеет знак, одинаковый со знаком y).

Это приводит нас к следующему выражению предиката наличия переполнения (помещающему результат в бит знака). Беззнаковый или знаковый сдвиг вправо на 31 бит дает нам результат в виде чисел 1/0 или -1/0.

$$x+y+c \qquad\qquad x-y-c$$

$$(x \equiv y) \,\&\, ((x+y+c) \oplus x) \qquad\qquad (x \oplus y) \,\&\, ((x-y-c) \oplus x)$$

$$((x+y+c) \oplus x) \,\&\, ((x+y+c) \oplus y) \qquad\qquad ((x-y-c) \oplus x) \,\&\, ((x-y-c) \equiv y)$$

Если взять вторую формулу для суммы и первую формулу для разности (не содержащие операции *эквивалентности),* то при использовании базового множества RISC-команд для проверки на переполнение потребуется выполнить три команды в дополнение к тем, которые вычисляют саму сумму (или разность). Четвертой может стать команда перехода к коду обработки переполнения.

Если при возникновении переполнения генерируется прерывание, то может потребоваться проверка, не вызовет ли сложение или вычитание переполнения (причем сама проверка ни в коем случае не должна вызвать переполнения). Для этого можно воспользоваться приведенными ниже формулами (в которых нет команд ветвления и которые заведомо не генерируют переполнения).

$$x+y+c \qquad\qquad x-y-c$$

$$z \leftarrow (x \equiv y) \,\&\, \mathtt{0x80000000} \qquad\qquad z \leftarrow (x \oplus y) \,\&\, \mathtt{0x80000000}$$

$$((x+y+c) \oplus x) \,\&\, ((x \oplus z) + y + c) \equiv y \qquad\qquad (x \oplus y) \,\&\, ((x \oplus z) - y - c) \oplus y$$

Присваивание в левом столбце устанавливает значение z равным 0x80000000, если знаки x и y одинаковы, и 0 в противном случае. Таким образом, во втором выражении слагаемые имеют противоположные знаки и переполнение возникнуть не может. Если и x и y неотрицательны, значение знакового бита в результате во втором выражении будет равным 1 тогда и только тогда, когда $(x - 2^{31}) + y + c \ge 0$, т.е. если $x + y + c \ge 2^{31}$, что и представляет собой условие возникновения переполнения при вычислении $x+y+c$. Если и x и y отрицательны, то значение бита знака во втором выражении будет равным 1 тогда и только тогда, когда $(x + 2^{31}) + y + c < 0$, т.е. если $x + y + c < -2^{31}$, что также представляет собой условие

переполнения при вычислении $x+y+c$. Член $x \equiv y$ гарантирует корректность результата (равный 0 знаковый бит), если x и y имеют разные знаки. Условия переполнения для разности (приведенные в правом столбце) получаются аналогично. Для реализации этих формул понадобится выполнить девять команд из базового множества RISC.

Может показаться, что использование значения разряда переноса дает эффективный алгоритм проверки на переполнение в случае знакового сложения, но на самом деле это не так. Однако есть еще один метод.

Если x — целое знаковое число, то $x+2^n$ представляет собой корректное беззнаковое число, которое получается в результате инвертирования старшего бита x. При сложении положительных знаковых чисел переполнение возникает тогда, когда $x+y \geq 2^{31}$, т.е. если $(x+2^{31})+(y+2^{31}) \geq 3 \cdot 2^{31}$. Последнее выражение означает, что при сложении беззнаковых операндов был перенос (так как сумма не меньше 2^{32}) и старший бит суммы равен 1. Аналогично, при сложении отрицательных чисел переполнение возникает тогда, когда значение разряда переноса и значение старшего разряда суммы равны 0.

Это дает нам следующий алгоритм обнаружения переполнения при знаковом сложении.

Вычисляем $(x \oplus 2^{31})+(y \oplus 2^{31})$ и получаем сумму s и значение бита переноса c.

Переполнение возникает тогда и только тогда, когда значение разряда переноса c равно значению старшего разряда суммы s.

Полученная сумма представляет собой корректное значение знакового сложения, поскольку инвертирование старшего бита обоих операндов не изменяет их суммы.

При вычитании используется такой же алгоритм, с тем отличием, что на первом шаге вместо суммы вычисляется разность. Мы считаем, что перенос в данном случае генерируется при вычислении $x-y$ как $x+\bar{y}+1$. Полученная разность представляет собой корректный результат знакового вычитания.

Несмотря на то что рассмотренные методы крайне интересны, на многих компьютерах они работают не так эффективно, как методы без использования переноса (например, вычисление переполнения как $(x \equiv y) \& (s \oplus x)$ для сложения и $(x \oplus y) \& (d \oplus x)$ для вычитания, где s и d — соответственно сумма и разность x и y).

Установка переполнения при знаковом сложении и вычитании

Зачастую переполнение при сложении знаковых операндов устанавливается посредством правила: "перенос в знаковый разряд отличается от переноса из знакового разряда". Как ни удивительно, но эта логика позволяет корректно определять переполнение как при сложении, так и при вычитании, считая, что разность $x-y$ вычисляется как $x+\bar{y}+1$. Более того, это утверждение корректно независимо от того, был ли перенос или заем либо нет. Конечно, то, что перенос в знаковый разряд легко вычислить, вовсе не означает, что существуют эффективные методы определения знакового переполнения. В случае сложения и вычитания перенос/заем формируется в знаковом разряде после вычисления приведенных ниже выражений (здесь c равно 0 или 1).

Перенос:	Заем:
$(x+y+c) \oplus x \oplus y$	$(x-y-c) \oplus x \oplus y$

В действительности для каждого разряда i эти выражения дают перенос/заем в разряд i.

Беззнаковое сложение/вычитание

При беззнаковом сложении/вычитании для вычисления предикатов переполнения можно использовать следующий метод без переходов, результат работы которого помещается в знаковый разряд. Методы, использующие команды сдвига вправо, эффективно работают только тогда, когда известно, что $c = 0$. Выражения в скобках вычисляют перенос или заем, который появляется из младшего разряда.

Беззнаковое вычисление $x + y + c$:

$$(x \,\&\, y) \mid ((x \mid y) \,\&\, \neg(x + y + c))$$

$$\left(x \overset{u}{\gg} 1\right) + \left(y \overset{u}{\gg} 1\right) + \left[\left(\left(x \,\&\, y\right) \mid \left(\left(x \mid y\right) \,\&\, c\right)\right) \,\&\, 1\right]$$

Беззнаковое вычисление $x - y - c$:

$$(\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x - y - c))$$

$$(\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y - c))$$

$$\left(x \overset{u}{\gg} 1\right) - \left(y \overset{u}{\gg} 1\right) - \left[\left(\left(\neg x \,\&\, y\right) \mid \left(\left(\neg x \mid y\right) \,\&\, c\right)\right) \,\&\, 1\right]$$

В [46] для беззнаковых *сложения* и *вычитания* есть существенно более простые формулы, использующие команды сравнения. При беззнаковом сложении переполнение (перенос) возникает, если полученная сумма меньше (при беззнаковом сравнении) любого из операндов. Эта и подобные формулы приведены ниже. К сожалению, нет возможности обобщить эти формулы для использования в них переменной *c*, которая представляет собой перенос или заем. Вместо этого приходится сначала проверять значение *c*, а затем, в зависимости от того, равно оно 0 или 1, использовать одно из следующих типов сравнений (напомним — вычисления сумм и разностей беззнаковые).

$x+y$	$x+y+1$	$x-y$	$x-y-1$
$\neg x \overset{u}{<} y$	$\neg x \overset{u}{\le} y$	$x \overset{u}{<} y$	$x \overset{u}{\le} y$
$x + y \overset{u}{<} x$	$x + y + 1 \overset{u}{\le} x$	$x - y \overset{u}{>} x$	$x - y - 1 \overset{u}{\ge} x$

В каждом из приведенных случаев перед выполнением сложения/вычитания вычисляется первая формула, которая проверяет, возможно ли переполнение, не вызывая при этом самого переполнения. Вторая формула вычисляется после выполнения сложения/вычитания, при которых возможно переполнение.

Для случая знакового сложения/вычитания аналогичного (с использованием сравнений) простого метода проверки, похоже, не существует.

Умножение

При умножении переполнение означает, что результат не помещается в 32 разрядах (результат как знакового, так и беззнакового умножения 32-битовых чисел всегда может быть представлен в виде 64-битового числа). Если доступны старшие 32 разряда произведения, то определить, имеет ли место переполнение, очень просто. Пусть $\mathrm{hi}(x \times y)$ и $\mathrm{lo}(x \times y)$ — старшая и младшая половины 64-битового произведения. Тогда условия переполнения можно записать следующим образом [46].

Беззнаковое произведение $x \times y$:

$$\mathrm{hi}(x \times y) \neq 0$$

Знаковое произведение $x \times y$:

$$\mathrm{hi}(x \times y) \neq \left(\mathrm{lo}(x \times y) \overset{s}{\gg} 31\right)$$

Один из способов проверки наличия переполнения состоит в контрольном делении. Однако при этом необходимо следить, чтобы не произошло деления на 0, а кроме того, при знаковом умножении возникают дополнительные сложности.

Переполнение при умножении имеет место, если приведенные ниже выражения истинны.

Беззнаковое умножение:

$$z \leftarrow x * y$$
$$y \neq 0 \mathbin{\bar{\&}} z \overset{u}{\div} y \neq x$$

Знаковое умножение:

$$z \leftarrow x * y$$
$$\left(y < 0 \mathbin{\&} x = -2^{31}\right) \mid \left(y \neq 0 \mathbin{\bar{\&}} z \div y \neq x\right)$$

Затруднения возникают при $x = -2"$ и при $y = -1$. В этом случае при умножении будет переполнение, но результат может получиться равным $-2^{"}$. Такой результат приводит к переполнению при делении, и таким образом оказывается возможным любой результат (на некоторых машинах). Поэтому данный частный случай должен проверяться отдельно, что и делается добавлением члена $\left(y < 0 \mathbin{\&} x = -2^{31}\right)$. Чтобы в приведенных выше выражениях предотвратить деление на 0, используется оператор "условного *и*" (в языке С — оператор `&&`).

Можно также использовать проверку делением без выполнения самого умножения (т.е. без генерации переполнения). При беззнаковом умножении переполнение возникает тогда и только тогда, когда $xy > 2^{32} - 1$, или $x > \left(\left(2^{32} - 1\right)/y\right)$, или, поскольку x — целое число, $x > \left\lfloor \left(2^{32} - 1\right)/y \right\rfloor$. В обозначениях компьютерной арифметики это выглядит следующим образом: $y \neq 0 \mathbin{\bar{\&}} x \overset{u}{>} \left(\text{0xFFFFFFFF} \overset{u}{\div} y\right)$.

Для целых знаковых чисел переполнение в результате умножения x и y определить сложнее. Если операнды имеют одинаковые знаки, то переполнение будет тогда и только тогда, когда $xy > 2^{31} - 1$. Если x и y имеют противоположные знаки, то переполнение будет тогда и только тогда, когда $xy < -2^{31}$. Эти условия могут быть проверены с помощью знакового деления (табл. 2.2).

Таблица 2.2. **Проверка на переполнение произведения знаковых операндов**

	$y > 0$	$y \leq 0$
$x > 0$	$x >$ 0x7FFFFFFF $\div\, y$	$y <$ 0x80000000 $\div\, x$
$x \leq 0$	$x <$ 0x80000000 $\div\, y$	$x \neq 0 \mathbin{\bar{\&}} y <$ 0x7FFFFFFF $\div\, x$

Как видно из таблицы, при проверке необходимо учесть четыре возможных случая. Вследствие проблем переполнения и представления числа $+2^{31}$ с объединением этих выражений могут возникнуть трудности.

В случае доступности беззнакового деления тест можно упростить. Если использовать абсолютные значения x и y, которые корректно представимы в виде целых чисел без знака, то весь тест можно записать так, как показано ниже. Переменная с имеет значение $2" - 1$, если x и y имеют одинаковые знаки, и $2^{м}$ в противном случае.

$$c \leftarrow \left((x \equiv y) \overset{s}{\gg} 31\right) + 2^{31}$$
$$x \leftarrow \mathrm{abs}(x)$$

$$y \leftarrow \text{abs}(y)$$

$$y \ne 0 \mathbin{\&} x \overset{u}{>} \left\lfloor x \overset{u}{\div} y \right\rfloor$$

Чтобы выяснить, будет ли переполнение при вычислении произведения $x * y$, можно использовать команду, вычисляющую *количество ведущих нулей у* операндов. Способ, основанный на использовании этой функции, позволяет более точно определить условие переполнения. Сначала рассмотрим умножение беззнаковых чисел. Легко показать, что если x и y — 32-разрядные величины, содержащие m и n ведущих нулей соответственно, то в 64-разрядном произведении будет $m+n$ или $m+n+\backslash$ ведущих нулей (или 64, если $x=0$ или $y=0$). Переполнение возникает тогда, когда в 64-разрядном произведении количество ведущих нулей оказывается меньше 32. Из этого следует:

если $\text{nlz}(x)+\text{nlz}(y) \ge 32$, переполнения точно не будет;
если $\text{nlz}(x)+\text{nlz}(y) < 30$, переполнение точно будет.

Если же $\text{nlz}(x)+\text{nlz}(y)=31$, то переполнение может быть, а может и не быть. Чтобы узнать, будет ли переполнение в этом частном случае, вычисляется значение $t = x\lfloor y/2 \rfloor$ (переполнения при этом вычислении не возникает). Поскольку xy равно 2/ (или $2t+x$, если y нечетно), переполнение при вычислении xy будет в том случае, когда $t \ge 2^{31}$. Все это приводит к коду, показанному в листинге 2.2, который вычисляет значение произведения xy с проверкой переполнения (в этом случае выполняется переход к метке `overflow`).

Листинг 2.2. Беззнаковое умножение с проверкой переполнения

```
unsigned x, y, z, m, n, t;

m = nlz(x);
n = nlz(y);
if (m + n <= 30) goto overflow;
t = x* (y >> 1);
if ((int)t < 0) goto overflow;
z = t * 2;
if (y & 1) {
    Z = z + X;
    if (z < x) goto overflow;
}
// В z содержится произведение x и y.
```

При знаковом умножении условие переполнения можно определить через количество ведущих нулей положительных аргументов и количество ведущих единиц отрицательных аргументов. Пусть $m = \text{nlz}(x)+\text{nlz}(\bar{x})$ и $n = \text{nlz}(y)+\text{nlz}(\bar{y})$, тогда:

если $m+n > 34$, переполнения точно не будет;
если $m+n < 31$, переполнение точно будет.

Остались два неоднозначных случая — 32 и 33. В случае $m+n=33$ переполнение возникает только тогда, когда оба аргумента отрицательны и произведение равно в точности 2^{31} (машинный результат равен -2^{31}), так что распознать эту ситуацию можно при помощи проверки корректности знака произведения (т.е. переполнение будет, если $m \oplus n \oplus (m * n) < 0$). Для случая $m+n=32$ определить наличие переполнения не так просто.

Не будем больше задерживаться на этой теме; заметим только, что определение наличия переполнения может основываться на значении $\text{nlz}(\text{abs}(x)) + \text{nlz}(\text{abs}(y))$, но и при этом возможны неоднозначные случаи, когда значение данного выражения равно 31 или 32.

Деление

При знаковом делении $x \div y$ переполнение возникает, если истинно выражение

$$y = 0 \mid (x = \mathbf{0x80000000} \,\&\, y = -1).$$

Большинство компьютеров сообщают о переполнении (или генерируют прерывание) при возникновении неопределенности типа $0 \div 0$.

Простой код для вычисления этого выражения, включая завершающую команду перехода к коду обработки переполнения, состоит из семи команд (три из которых являются командами перехода). Создается впечатление, что этот код улучшить нельзя, однако рассмотрим некоторые возможности для этого.

$$\left[\text{abs}(y \oplus \mathbf{0x80000000}) \mid (\text{abs}(x) \,\&\, \text{abs}(y = \mathbf{0x80000000}))\right] < 0$$

Здесь сначала вычисляется выражение в скобках, а затем выполняется команда перехода, если полученный результат меньше 0. Это выражение на машине, имеющей все необходимые команды (включая сравнение с 0), вычисляется примерно за девять команд с учетом загрузки константы и завершающего перехода.

Еще один метод предполагает первоначальное вычисление значения

$$z \leftarrow (x \oplus \mathbf{0x80000000}) \mid (y + 1)$$

(что на большинстве машин выполняется за три команды), а затем выполнение проверки и ветвления по условию $y = 0 \mid z = 0$ одним из перечисленных способов.

$$((y \mid -y) \,\&\, (z \mid -z)) \geq 0$$

$$(\text{nabs}(y) \,\&\, \text{nabs}(z)) \geq 0$$

$$\left((\text{nlz}(y) \mid \text{nlz}(z)) \overset{u}{\gg} 5\right) \neq 0$$

Вычисление этих выражений может быть реализовано за девять, семь и восемь команд соответственно (при наличии всех необходимых для вычисления команд). Для PowerPC наиболее эффективным оказывается последний из перечисленных методов.

При беззнаковом делении переполнение возникает тогда и только тогда, когда $y = 0$.

2.13. Флаги условий после сложения, вычитания и умножения

Во многих процессорах в результате выполнения арифметических действий устанавливаются так называемые флаги условий, в совокупности представляющие слово состояния процессора. Зачастую имеется только одна команда *сложения,* после которой выполняется оценка результата для беззнаковых или знаковых операндов (но не для смешанных типов). Как правило, на флаги условий оказывают влияние следующие события:

- был ли перенос или нет (беззнаковое переполнение);
- было или нет знаковое переполнение;
- является ли полученный 32-битовый результат, интерпретируемый как знаковое целое число в дополнительном коде без учета переноса и переполнения, отрицательным, положительным или нулевым.

На некоторых старых моделях машин в случае переполнения обычно выдается сообщение о том, какой именно результат (в случае *сложения* и *вычитания* — 33-битовый) получен: положительный, отрицательный или нулевой. Однако использовать такое сообщение в компиляторах языков высокого уровня — непростая задача, так что данная возможность нами не рассматривается.

При сложении возможны только 9 из 12 комбинаций этих трех событий. Невозможны следующие три комбинации: "нет переноса, переполнение, результат > 0", "нет переноса, переполнение, результат = 0" и "перенос, переполнение, результат < 0". Таким образом, для флагов условий требуется четыре бита. Две из комбинаций флагов уникальны в том смысле, что могут произойти только при единственном значении исходных аргументов: комбинация "нет переноса, нет переполнения, результат = 0" возникает только при сложении 0 с 0; комбинация "перенос, переполнение, результат = 0" возникает только при сложении двух максимальных отрицательных чисел. Эти замечания справедливы и при наличии "входного переноса", т.е. при вычислении $x + y + 1$.

В случае вычитания полагаем, что для вычисления разности $x - y$ компьютер вычисляет сумму $x + \bar{y} + 1$ с переносом, который образуется так же, как и при *сложении* (понятие "перенос" при вычитании трактуется наоборот: перенос, равный 1, означает, что результат помещается в одно слово, а перенос, равный 0, означает, что результат в одно слово не помещается). Следовательно, при вычитании могут произойти только семь из всех возможных комбинаций событий. Невозможны те же три комбинации, что и при сложении, а кроме того, еще две комбинации: "нет переноса, нет переполнения, результат = 0" и "перенос, переполнение, результат = O".

Если умножение на компьютере может давать произведение двойной длины, то желательно иметь две команды умножения: одну для знаковых операндов, а другую — для беззнаковых. (На 4-разрядной машине в результате умножения знаковых операндов произведение **F×F** = 01, а в случае перемножения беззнаковых операндов — **F×F** = El (в шестнадцатеричной записи)). Ни переноса, ни переполнения при выполнении этих команд умножения не возникает, так как результат умножения всегда может разместиться в двойном слове.

Если же произведение дает в результате одно (младшее) слово, то в случае, когда операнды и результат интерпретируются как целые беззнаковые числа, перенос означает, что результат не помещается в одно слово. Если же операнды и результат интерпретируются как знаковые целые числа в дополнительном коде, то, если результат не помещается в слово, устанавливается флаг переполнения. Таким образом, при умножении возможны только девять комбинаций событий. Невозможны комбинации "нет переноса, переполнение, результат > O", "нет переноса, переполнение, результат = 0" и "перенос, нет переполнения, результат = O". Таким образом, при рассмотрении сложения, вычитания и умножения получаем, что реально могут встретиться только 10 комбинаций флагов условий из 12 возможных.

2.14. Циклический сдвиг

Здесь все тривиально. Приведенные ниже выражения справедливы для любых и от O до 32 включительно, даже если сдвиги выполняются по модулю 32.

Циклический сдвиг влево на и разрядов: $y \leftarrow \left(x \ll n\right) \mid \left(x \gg (32 - n)\right)$

Циклический сдвиг вправо на *n* разрядов: $y \leftarrow \left(x \gg n\right) \mid \left(x \ll (32 - л)\right)$

2.15. Сложение/вычитание двойных слов

Взяв за основу любое из приведенных в подразделе "Беззнаковое сложение/вычитание" раздела 2.12 выражение для переполнения при беззнаковом сложении и вычитании, можно легко реализовать сложение и вычитание двойных слов, не используя при этом значение бита переноса. Пусть (z_1, z_0) — результат сложения операндов (x_1, x_0) и (y_1, y_0). Индекс 1 обозначает старшее полуслово, а индекс 0 — младшее. Считаем также, что при сложении используются все 32 бита регистров. Младшие слова интерпретируются как беззнаковые величины. Тогда

$$
\begin{aligned}
&z_0 \leftarrow x_0 + y_0 \\
&c \leftarrow \left[(x_0 \,\&\, y_0) \,|\, ((x_0 \,|\, y_0) \,\&\, \neg z_0) \right] \overset{u}{\gg} 31 \\
&z_1 \leftarrow x_1 + y_1 + c.
\end{aligned}
$$

Всего для вычисления потребуется девять команд. Во второй строке можно использовать выражение $c \leftarrow \left(z_0 \overset{u}{<} x_0 \right)$, допускающее получение конечного результата за четыре команды на машинах, в наборе команд которых имеется оператор сравнения в форме, помещающей результат (0 или 1) в регистр, как, например, команда SLTU *(Set on Less Then Unsigned— беззнаковая проверка "меньше, чем")* в MIPS [46].

Аналогичный код используется для вычисления разности двойных слов.

$$
\begin{aligned}
&z_0 \leftarrow x_0 - y_0 \\
&b \leftarrow \left[(\neg x_0 \,\&\, y_0) \,|\, ((x_0 \equiv y_0) \,\&\, z_0) \right] \overset{u}{\gg} 31 \\
&z_1 \leftarrow x_1 - y_1 - b
\end{aligned}
$$

На машинах с полным набором логических команд для вычисления потребуется выполнить восемь команд. Если вторую строку записать как $b \leftarrow \left(x_0 \overset{u}{<} y_0 \right)$, то на машинах, имеющих команду SLTU, для вычисления разности понадобится всего четыре команды.

На большинстве машин сложение и вычитание двойных слов реализуется за пять команд, посредством представления данных в младшем слове с помощью 31 бита (старший бит всегда равен 0, за исключением временного хранения в нем бита переноса или заема).

2.16. Сдвиг двойного слова

Пусть (x_1, x_0) — два 32-разрядных слова, которые требуется сдвинуть вправо или влево как единое 64-разрядное слово, в котором $x\backslash$ представляет старшее полуслово, и пусть (y_1, y_0) представляет собой аналогично интерпретируемый результат сдвига. Будем считать, что величина сдвига я может принимать любые значения от 0 до 63. Предположим также, что машина может выполнять команду сдвига по модулю 64 или большему. Это означает, что если величина сдвига лежит в диапазоне от 32 до 63 или от -32 до -1, то в результате сдвига получится слово, состоящее из одних нулевых битов. Исключением является команда знакового сдвига вправо, так как в этом случае 32 разряда результата будут заполнены значением знакового разряда слова. (Рассматриваемый алгоритм не работает на процессорах Intel x86, сдвиг в которых выполняется по модулю 32.)

2.18. Функции Doz, Max, Min

Функция doz для знаковых аргументов определяется следующим образом:

$$\text{doz}(x, y) = \begin{cases} x - y, & x \geq y, \\ 0, & x < y. \end{cases}$$

Функцию doz называют также "вычитанием в первом классе", так как, если вычитаемое больше уменьшаемого, результат равен 0. Эту функцию удобно использовать при вычислении двух других функций — $\max(x, y)$ и $\min(x, y)$. В связи с этим следует отметить, что функция $\text{doz}(x, y)$ может быть и отрицательной, если при вычислении разности произошло переполнение. В Fortran эта функция используется в реализации функции IDIM, хотя в этом языке программирования при возникновении переполнения результат считается неопределенным.

Похоже, что очень хорошей реализации функций doz, max и min без использования команд перехода, применимой на большинстве компьютеров, просто не существует. Пожалуй, лучшее, что можно сделать, — это вычислить $\text{doz}(x, y)$ с использованием одного из выражений для рассматривавшегося в этой главе предиката $x < y$, а затем вычислить значения функций max и min.

$$\begin{aligned}
&d \leftarrow x - y \\
&\text{doz}(x, y) = d \mathbin{\&} \left[\left(d \equiv \left((x \oplus y) \mathbin{\&} (d \oplus x) \right) \right) \overset{s}{\gg} 31 \right] \\
&\max(x, y) = y + \text{doz}(x, y) \\
&\min(x, y) = x - \text{doz}(x, y)
\end{aligned}$$

Вычислить функцию $\text{doz}(x, y)$ можно за семь команд, если в наборе имеется команда *эквивалентности,* или за восемь команд в противном случае. Для вычисления функций max и min необходимо выполнить еще одну команду.

В случае беззнаковых аргументов используем беззнаковые версии указанных функций.

$$\begin{aligned}
&d \leftarrow x - y \\
&\text{dozu}(x, y) = d \mathbin{\&} \neg\left[\left((\neg x \mathbin{\&} y) \mid \left((x \equiv y) \mathbin{\&} d \right) \right) \overset{s}{\gg} 31 \right] \\
&\text{maxu}(x, y) = y + \text{dozu}(x, y) \\
&\min(x, y) = x - \text{dozu}(x, y)
\end{aligned}$$

В компьютере IBM RS/6000 (и его предшественнике 801) есть отдельная команда для вычисления функции $\text{doz}(x, y)$, так что на этих машинах функции $\max(x, y)$ и $\min(x, y)$ с целыми знаковыми аргументами вычисляются за две команды, что весьма эффективно (непосредственная реализация этих функций требует больших затрат).

На машинах, имеющих команду *условной пересылки,* возможна двухкомандная реализация деструктивных[2] функций max и min. Например, при использовании нашего RISC-компьютера с расширенным набором команд выражение $x \leftarrow \max(x, y)$ можно вычислить следующим образом:

```
cmplt z, x, y    ; Установить z = 1, если x < y, иначе z = 0
movne x, z, y    ; Если z не равен 0, то присвоить x = y
```

---

[2] Деструктивные операции представляют собой операции, которые перезаписывают один или несколько своих аргументов.

2.19. Обмен содержимого регистров

Существует старый, хорошо известный способ обмена содержимым двух регистров без использования третьего [34].

$$
\begin{aligned}
&x \leftarrow x \oplus y\\
&y \leftarrow y®x\\
&x \leftarrow x \oplus y
\end{aligned}
$$

Этот метод хорошо работает на двухадресной машине. Он остается работоспособным и при замене оператора $\oplus$ дополнительным к нему оператором эквивалентности $\equiv$. Возможно также использование различных наборов операций сложения и вычитания.

$$
\begin{array}{lll}
x \leftarrow x + y & x \leftarrow x - y & x \leftarrow y - x\\
y \leftarrow x - y & y \leftarrow y + x & y \leftarrow y - x\\
x \leftarrow x - y & x \leftarrow y - x & x \leftarrow x + y
\end{array}
$$

К сожалению, в каждом из вариантов обмена есть команда, неприемлемая для двухадресной машины (если, конечно, среди команд машины нет "обратного вычитания").

Вот еще одна маленькая хитрость, которая может оказаться полезной в приложениях с двойной буферизацией, где меняются местами два указателя. Первая команда выносится за пределы цикла, в котором выполняется обмен (хотя при этом и теряется преимущество сохранения регистра).

$$
\begin{array}{rl}
\text{Вне цикла:} & t \leftarrow x\,\Phi\,y\\
\text{Внутри цикла:} & x \leftarrow x\,©\,t\\
 & y \leftarrow y \oplus t
\end{array}
$$

Обмен соответствующих полей регистров

Зачастую требуется обменять биты двух регистров x и y, если i-й бит маски $m_i = 1$, и оставить их неизменными, если $m_i = 0$. Под обменом "соответствующими" полями подразумевается обмен без сдвигов. В маске m единичные биты не обязательно должны быть смежными. Приведем простейший метод решения поставленной задачи.

$$
\begin{aligned}
&x' \leftarrow (x \,\&\, \bar{m}) \,|\, (y \,\&\, m)\\
&y \leftarrow (y \,\&\, \bar{m}) \,|\, (x \,\&\, m)\\
&x \leftarrow x'
\end{aligned}
$$

Использование промежуточных временных данных в четырех выражениях с командой *и* требует выполнения семи команд при условии, что загрузка m или m занимает одну команду и имеется команда *и-не*. Если машина в состоянии вычислить четыре независимых команды *и* параллельно, то время выполнения равно всего трем тактам.

Более эффективный метод (пять команд, требующих для выполнения четыре такта на машине с неограниченными возможностями распараллеливания вычислений на уровне команд) показан ниже, в столбце (а). Этот алгоритм использует три команды *исключающего или*.

$$
\begin{array}{lll}
\text{(а)} & \text{(б)} & \text{(в)}\\
x \leftarrow x \oplus y & x \leftarrow x \equiv y & t \leftarrow (x \oplus y) \,\&\, m\\
y \leftarrow y \oplus (x \,\&\, m) & y \leftarrow y \equiv (x \,|\, \bar{m}) & x \leftarrow x \oplus t\\
x \leftarrow x \oplus y & x \leftarrow x \equiv y & y \leftarrow y \oplus t
\end{array}
$$

Метод, предложенный в столбце (б), лучше использовать в тех случаях, когда m не помещается в поле непосредственно задаваемого значения, но в него можно поместить m , а в наборе команд имеется команда *эквивалентности*.

Метод из столбца (в) предложен в [19]. Он тоже требует выполнения пяти команд (в предположении, что для загрузки от в регистр используется одна команда), но на машинах с достаточными возможностями распараллеливания вычислений на уровне команд выполнение этого кода занимает всего три такта.

Обмен двух полей одного регистра

Предположим, что в регистре x необходимо поменять два поля (одинаковой длины), не изменяя при этом прочие биты регистра — т.е. нам требуется поменять местами поля B и D в пределах одного машинного слова, не изменяя при этом содержимое полей A, C и E. Поля расположены друг от друга на расстоянии k бит.

	← k →

x:	A	B	C	D	E

Следующий код сдвигает поля D и B на новые позиции и затем комбинирует полученные слова с помощью команд *и* и *или*.

$$t_1 = (x \,\&\, m) \ll k$$
$$t_2 = \left(x \overset{u}{\gg} k\right) \& \, m$$
$$x' = (x \,\&\, m') \,|\, t_1 \,|\, t_2$$

Здесь маска m содержит единичные биты в поле D (и нулевые биты в других полях), а маска m' содержит единичные биты в полях A, C и E. На машине с неограниченными возможностями распараллеливания вычислений на уровне команд требуется выполнение девяти команд за четыре такта, так как для загрузки двух масок необходимо выполнить две команды.

Ниже описан метод обмена полями [19], который в тех же предположениях требует выполнения семи команд и выполняется за пять тактов. Он аналогичен алгоритму обмена соответствующими полями двух регистров, описанному ранее в подразделе "Обмен соответствующих полей регистров" в столбце (в). Здесь, как и ранее, m — маска, выделяющая поле D.

$$t_1 = \left[x \oplus \left(x \overset{u}{\gg} k\right)\right] \& \, m$$
$$t_2 = t_1 \ll k$$
$$x' = x \oplus t_1 \oplus t_2$$

Идея состоит в том, что t_1 в позициях поля D содержит биты $B \oplus D$ (и нули в остальных полях), а t_2 содержит биты B ® D в поле B. Этот код, как и приведенный ранее, работает корректно, даже если B и D представляют собой поля, разбитые на части, т.е. даже если в маске от единичные биты не являются смежными.

Условный обмен

Методы обмена из предыдущих двух разделов, использующие команду *исключающего или*, вырождаются в отсутствие каких-либо действий, если все биты маски от нулевые. Следовательно, эти методы могут выполнять условный обмен содержимым регистров, соответствующих полей регистров или полей в одном регистре, если некоторое условие c истинно, и не выполнять его при ложности условия. Для этого достаточно, чтобы все биты маски m были нулевыми, если условие ложно, и име-

лись корректно установленные единичные биты при выполнении условия c. Если формирование маски m выполняется без команд ветвления, код условного обмена также не будет содержать этих команд.

2.20. Выбор среди двух или большего количества значений

Предположим, что переменная x может принимать только два возможных значения — a и b. Пусть требуется присвоить переменной значение, отличающееся от текущего, причем алгоритм не должен зависеть от конкретных значений a и b. Например, в компиляторе x может быть кодом одной из операций — *переход при выполнении условия* или *переход при невыполнении условия,* и вам требуется изменить текущий код на противоположный.

Вот простейший код переключения:

```
if (x == a) x = b;
else x = a;
```

А это его эквивалент в языке С:

```
x = (x == a) ? b : a;
```

Однако существенно лучше (по крайней мере эффективнее) использовать другой способ: $x \leftarrow a + b - x$ или $x \leftarrow a \oplus b \oplus x$. Если а и ft — константы, то вычисления требуют всего одной или двух команд из базового множества RISC-команд. Естественно, переполнение при вычислении $a + b$ можно игнорировать.

Возникает закономерный вопрос: существует ли эффективный метод циклического перебора трех и более значений? Иными словами, пусть заданы три произвольные, не равные друг другу константы a, b и с. Требуется найти легко вычисляемую функцию f, которая удовлетворяет условиям

$$f(a) = b; \quad f(b) = c; \quad f(c) = a.$$

Любопытно отметить, что всегда существует полиномиальная функция, удовлетворяющая указанным условиям. В случае трех констант это функция

$$f(x) = \frac{(x-a)(x-b)}{(c-a)(c-b)}a + \frac{(x-b)(x-c)}{(a-b)(a-c)}b + \frac{(x-c)(x-a)}{(b-c)(b-a)}c, \quad (4)$$

Идея заключается в том, что при $x = a$ первый и последний члены функции обращаются в нуль и остается только средний член при коэффициенте b и т.д. Для вычисления этой функции необходимо выполнить 14 арифметических операций, при этом в общем случае промежуточные результаты превышают размер машинного слова, хотя мы имеем дело всего лишь с квадратичной функцией. Однако если для вычисления полинома применить схему Горнера[3], то понадобится выполнить только пять арифметических операций (четыре для вычисления квадратов с целыми коэффициентами и заключительное деление). После преобразований (4) получим следующий вид функции:

---

[3] Правило Горнера состоит в вынесении x за скобки. Например, полином четвертой степени $ax^4 + bx^3 + cx^2 + dx + e$ вычисляется как $x(x(x(ax+b)+c)+d)+e$. Для вычисления полинома степени n требуется n умножений и n сложений; при этом наличие команды умножения со сложением существенно повышает эффективность вычислений.

$$f(x)=\frac{1}{(a-b)(a-c)(b-c)}\{[(a-b)a+(b-c)b+(c-a)c]x^2+$$
$$+[(a-b)b^2+(b-c)c^2+(c-a)a^2]x+$$
$$+[(a-b)a^2b+(b-c)b^2c+(c-a)ac^2]\}.$$

Полученное выражение слишком громоздко и непригодно для практического использования.

Еще один метод аналогичен формуле (4) в том смысле, что при вычислении также остается только один из трех членов функции:

$$f(x)=((-(x=c))\&a)+((-(x=a))\&b)+((-(x=b))\&c).$$

Для вычисления этого выражения, при условии наличия в наборе команд предиката равенства, требуется выполнить 11 команд, не считая команды загрузки констант. Так как две команды сложения суммируют два нулевых значения с одним ненулевым, их можно заменить командами *и* или *исключающего или.*

Данную формулу можно упростить, если предварительно вычислить значения $a-c$ и *Ь-с* [19]:

$$f(x)=((-(x=c))\&(a-c))+((-(x=a))\&(b-c))+c \text{ или}$$
$$f(x)=((-(x=c))\&(a\oplus c))\oplus((-(x=a))\&(b\oplus c))\oplus c.$$

Каждое из этих выражений вычисляется восемью командами. Но на большинстве компьютеров это, вероятно, ничуть не эффективнее простого кода на языке С, для выполнения которого требуется от четырех до шести команд для малых значений а, Ь и с.

```
if (x == a) x = b;
else if (x == Ь) x = c;
else x = a;
```

В [19] предлагается еще один оригинальный метод без переходов с циклом выборки из трех значений, работающий даже там, где нет команд сравнения. На большинстве машин его можно выполнить за восемь команд.

Пусть *а, Ь* и с не равны друг другу. Следовательно, имеются два разряда n_1 и n_2 такие, что биты чисел *а, b* и с в этих разрядах различны, и два числа, оба бита которых отличаются от битов другого числа. Проиллюстрируем это на примере чисел 21, 31 и 20.

1	0	1	0	1	c
1	1	1	1	1	a
1	0	1	0	0	*
	n_1			л,	

Без потери общности переименуем *а, Ъ* и *с* так, чтобы биты а и ***b*** были различны в обоих разрядах, как и показано выше. Тогда существует два возможных набора битов в позиции n_1, а именно $(a_{n_1},b_{n_1},c_{n_1})=(0,0,1)$ или $(1,0,0)$. Аналогично, для битов в позиции n_2 также возможны два варианта: $(a_{n_2},b_{n_2},c_{n_2})=(0,1,0)$ или $(1,0,1)$. Это приводит нас к рассмотрению четырех возможных случаев, формулы для которых приведены ниже.

Случай 1. $(a_{n_1},b_{n_1},c_{n_1})=(0,1,1)$, $(a_{n_2},b_{n_2},c_{n_2})=(0,1,0)$:

$$f(x)=x_{n_1}*(a-b)+x_{n_2}*(c-a)+b$$

Случай 2. $(a_{n_1}, b_{n_1}, c_{n_1}) = (0,1,1)$, $(a_{n_2}, b_{n_2}, c_{n_2}) = (1,0,1)$:

$$f(x) = x_{n_1} * (a - b) + x_{n_2} * (a - c) + (b + c - a)$$

Случай 3. $(a_n, b_n, c_n) = (1,0,0)$, $(a_{n_2}, b_{n_2}, c_{n_2}) = (0,1,0)$:

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (c - a) + a$$

Случай 4. $(a_{n_1}, b_{n_1}, c_{n_1}) = (1,0,0)$, $(a_{n_2}, b_{n_2}, c_{n_2}) = (1,0,1)$:

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (a - c) + c$$

В этих формулах левый операнд каждого умножения представляет собой отдельный бит. Умножение на 0 или 1 можно заменить операцией *и с 0* или с числом, все биты которого равны 1. Таким образом, эти формулы можно переписать иначе, например для первого случая получим:

$$f(x) = \left(\left(x \ll \left(31 - n_1\right)\right) \overset{s}{\gg} 31\right) \& \left(a - b\right) + \left(\left(x \ll \left(31 - n_2\right)\right) \overset{s}{\gg} 31\right) \& \left(c - a\right) + b\,.$$

Поскольку все переменные, кроме x, являются константами, функцию можно вычислить восемью базовыми RISC-командами. Как и ранее, команды сложения и вычитания можно заменить командами *исключающего или.*

Данная идея может быть распространена на случай циклической выборки из четырех и более констант. Главное — найти такие номера битов n_1, n_2, ..., которые бы идентифицировали константы единственным образом. Для четырех констант всегда достаточно трех позиций битов. Затем (в случае четырех констант) решаются уравнения для *s, t, u* и v (т.е. решается система из четырех линейных уравнений, в которой $f(x)$ принимает значения *a, b, c* и *d,* а коэффициенты x_{n_i} равны 0 или 1):

$$f(x) = x_{n_1} s + x_{n_2} t + x_{n_3} u + v\,.$$

Если все четыре константы определяются двумя битами единственным образом, то уравнение принимает следующий вид:

$$f(x) = x_{n_1} s + x_{n_2} t + x_{n_1} x_{n_2} u + v\,.$$

ГЛАВА 3

ОКРУГЛЕНИЕ К СТЕПЕНИ 2

3.1. Округление к кратному степени 2

Округление целого беззнакового числа * в меньшую сторону, например к ближайшему числу, кратному 8, тривиально: для этого можно использовать выражение x & -8 или $\left(x \overset{u}{\gg} 3\right) \ll 3$. Этот способ работает и со знаковыми целыми числами, просто здесь "округление в меньшую сторону" означает округление к меньшему отрицательному числу, например: (-37) & $(-8) = -40$.

Округление в большую сторону выполняется практически так же просто. Например, округление беззнакового целого x в большую сторону к следующему кратному 8 реализуется одной из формул: $(x+7)$ & -8 или $x + (-x \,\&\, 7)$.

Эти формулы корректны и для знаковых целых чисел, причем "округление в большую сторону" означает округление в положительном направлении. Второе слагаемое во втором выражении может быть полезно само по себе, так как означает разность между ближайшим кратным 8 числом, большим x, и самим x [22].

Для округления знаковых целых чисел к ближайшему кратному 8 по направлению к 0 (т.е. -37 округляется до -32) можно просто скомбинировать приведенные ранее выражения:

$$t \leftarrow \left(x \overset{s}{\gg} 31\right) \,\&\, 7;$$
$$(x+t) \,\&\, -8.$$

Первую формулу можно переписать несколько иначе: $t \leftarrow \left(x \overset{s}{\gg} 2\right) \overset{u}{\gg} 29$, что может пригодиться при округлении на компьютере с отсутствующей командой *и с непосредственно заданным значением* или в том случае, если константа оказывается слишком велика для размещения в отводимом для нее поле.

Иногда при округлении указывается множитель округления, представляющий собой $\log_2$ от величины округления (так, значение 3 означает округление к числу, кратному 8, так как $\log_2 8 = 3$). В этом случае можно использовать приведенные ниже формулы, где k — множитель округления.

Округление к меньшему числу: $x \,\&\, ((-1) \ll k)$

$\left(x \overset{u}{\gg} k\right) \ll k$

Округление к большему числу: $t \leftarrow (1 \ll k) - 1;$ $(x+t) \,\&\, \neg t$ или

$t \leftarrow (-1) \ll k;$ $(x - t - 1) \,\&\, t$

3.2. Округление к ближайшей степени 2

Определим функции $\mathrm{flp2}(x)$ и $\mathrm{clp2}(x)$, которые округляют число x к ближайшей целой степени 2, следующим образом:

$$\mathrm{flp2}(x)=\begin{cases}\text{не определена}, & x<0,\\ 0, & x=0,\\ 2^{\lfloor \log_2 x\rfloor}, & x>0;\end{cases}\qquad \mathrm{clp2}(x)=\begin{cases}\text{не определена}, & x<0,\\ 0, & x=0,\\ 2^{\lceil \log_2 x\rceil}, & x>0.\end{cases}$$

Значение функции $\mathrm{flp2}(x)$ представляет собой наибольшую целую степень 2, не превосходящую значение x, а функции $\mathrm{clp2}(x)$ — наименьшую целую степень 2, не меньшую, чем x. Эти определения имеют смысл и для нецелых x (например, $\mathrm{flp2}(0.1)=0.0625$). Эти функции удовлетворяют ряду отношений, аналогичных отношениям между функциями $\mathrm{floor}(x)\equiv\lfloor x\rfloor$ и $\mathrm{ceil}(x)\equiv\lceil x\rceil$, в частности приведенным ниже.

$\lfloor x\rfloor=\lceil x\rceil$, только если x — целое	$\mathrm{flp2}(x)=\mathrm{clp2}(x)$, только если x — степень 2 или 0
$\lfloor x+n\rfloor=\lfloor x\rfloor+n$	$\mathrm{flp2}(2^n x)=2^n\,\mathrm{flp2}(x)$
$\lceil x\rceil=-\lfloor -x\rfloor$	$\mathrm{clp2}(x)=1/\mathrm{flp2}(1/x),\ x\neq 0$

При вычислениях используются только целые беззнаковые значения x, так что обе функции определены для любых возможных значений x. Кроме того, используется требование, чтобы вычисленное значение представляло собой значение по модулю 2^{32} (таким образом, $\mathrm{clp2}(x)=0$ при $x>2^{31}$). Далее представлены значения функций для некоторых значений x.

x	$\mathrm{flp2}(x)$	$\mathrm{clp2}(x)$
0	0	0
1	1	1
2	2	2
3	2	4
4	4	4
5	4	8
...	...	...
$2^{31}-1$	2^{30}	2^{31}
2^{31}	2^{31}	2^{31}
$2^{31}+1$	2^{31}	0
...	...	...
$2^{32}-1$	2^{31}	0

Функции $\mathrm{flp2}(x)$ и $\mathrm{clp2}(x)$ связаны между собой определенными соотношениями, которые позволяют вычислить значения одной функции из значений другой (с учетом указанных ограничений).

$$\mathrm{clp2}(x)=\begin{cases}2\,\mathrm{flp2}(x-1), & x\neq 1,\\ \mathrm{flp2}(2x-1), & 1\le x\le 2^{31},\end{cases}\qquad \mathrm{flp2}(x)=\begin{cases}\mathrm{clp2}(x\overset{u}{\div}2+1), & x\neq 0,\\ \mathrm{clp2}(x+1)\overset{u}{\div}2, & x<2^{31}.\end{cases}$$

Функции округления в большую и меньшую сторону можно достаточно легко вычислить с помощью функции, вычисляющей *количество ведущих нулевых битов,* как показано ниже. Однако, чтобы эти формулы можно было использовать для x=0 и $x>2^{31}$, команды сдвига на вашем компьютере должны давать 0 при сдвиге на величины -1, 32 и 63. На многих машинах (например, PowerPC) есть команда "сдвига по модулю 64", которая работает именно так. Отрицательный сдвиг эквивалентен сдвигу в противоположном направлении (т.е. сдвиг влево на -1 — это просто сдвиг вправо на 1).

$$
\begin{aligned}
\text{flp2}(x) &= 1 \ll (31 - \text{nlz}(x)) = \\
&= 1 \ll (\text{nlz}(x) \oplus 31) = \\
&= 0\text{x}80000000 \overset{u}{\gg} \text{nlz}(x) \\
\text{clp2}(x) &= 1 \ll (32 - \text{nlz}(x-1)) = \\
&= 0\text{x}80000000 \overset{u}{\gg} (\text{nlz}(x-1) - 1)
\end{aligned}
$$

Округление в меньшую сторону

В листинге 3.1 приведен алгоритм без ветвления, который полезен в случае отсутствия функции, вычисляющей *количество ведущих нулевых битов.* Алгоритм основан на распространении вправо крайнего слева единичного бита и требует для реализации выполнения 12 команд.

Листинг 3.1. Наибольшая степень 2, не превосходящая значение *x*

```
unsigned flp2(unsigned x)
{
    x = x | (x >> 1);
    x = x | (x >> 2);
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x | (x >> 16);
    return x - (x >> 1);
}
```

В листинге 3.2 показаны два простых цикла, вычисляющих одну и ту же функцию. Все переменные в приведенных фрагментах представляют собой беззнаковые целые числа. Цикл справа обнуляет крайний справа единичный бит переменной *x* до тех пор, пока она не становится равной 0, после чего возвращается предыдущее значение *x*.

Листинг 3.2. Округление *x* в меньшую сторону

```
y = 0x80000000;                do {
while (y > x)                     y = x;
    y = y >> 1;                   x = x & (x - 1);
return y;                      } while(x != 0);
                               return y;
```

При вычислении левого фрагмента требуется $4\text{nlz}(x)+3$ команды; цикл справа при $x \neq 0$ требует выполнения $4\text{pop}(x)$ команд[1], если сравнение с 0 имеет нулевую стоимость.

[1] Функция $\text{pop}(x)$ возвращает количество единичных битов в x.

Округление в большую сторону

Использование распространения старшего бита вправо дает хороший алгоритм для округления к ближайшей большей степени 2. Представленный в листинге 3.3 алгоритм не имеет команд ветвления и выполняется за 12 команд.

Листинг 3.3. Наименьшая степень 2, не уступающая значению *x*

```
unsigned clp2(unsigned x)
{
    x = x - 1;
    x = x | (x >> 1);
    x = x | (x >> 2);
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x | (x >> 16);
    return x + 1;
}
```

Попытка провести вычисление с использованием цикла работает не так хорошо, как хотелось бы.

```
y = 1;
while (y < x)    // Беззнаковое сравнение
    y = 2*y;
return y;
```

Для $x=0$ данный алгоритм дает 1, что не совсем то, что требуется; а при $x \geq 2^{31}$ происходит зацикливание. Вычисления требуют выполнения $4я+3$ команд, где и — степень 2 возвращаемого кодом числа. Таким образом, этот алгоритм будет работать медленнее алгоритма без ветвления уже при $n \geq 3$ (т.е. при $x \geq 8$).

3.3. Проверка пересечения границы степени 2

Предположим, что вся память разделена на блоки, причем размер каждого блока представляет собой степень 2. Нумерация адресов начинается с 0. Блоки могут быть словами, двойными словами, страницами и т.д. Пусть дан некоторый начальный адрес *a* и длина /. Требуется определить, пересекает ли диапазон адресов от a до $a+l-1, l \geq 2$, границы блока или нет (величины a и l — целые беззнаковые, полностью размещаемые в регистрах).

Если /=0 или 1, то, независимо от значения *a,* пересечения границ блока не будет. Если / превышает размер блока, то, каким бы ни было значение *a,* произойдет выход за границы блока. Для очень больших значений / (с возможным циклическим возвратом) выход за границы блока может произойти, даже если первый и последний байты диапазона адресов расположены в одном блоке.

Для IBM System/370 определить, произойдет ли выход за границы блока, оказывается на удивление просто [7]. Вот как этот метод работает для размера блока, равного 4096 байт (стандартный размер страницы).

```
O      RA, =A(-4096)
ALR    RA,RL
BO     CROSSES
```

В первой команде над регистром RA (в котором содержится начальный адрес a) и числом 0xFFFFF000 выполняется команда *побитового или.* Вторая команда добавляет к

регистру длину, в результате чего устанавливается двухбитовый флаг условия. Первый флаг устанавливается в 1, если при сложении произошел перенос, а второй оказывается установленным, если 32-битовый результат сложения ненулевой. Переход в последней команде выполняется только тогда, когда оба флага условия равны 1. После выполнения перехода RA будет содержать длину диапазона адресов, выходящую за пределы первой страницы (дополнительная возможность, которая не запрашивалась).

Если, например, $a=0$ и $l=4096$, перенос будет иметь место, однако результат в регистре RA окажется нулевым, так что перехода к метке CROSSES *не* будет.

Теперь посмотрим, нельзя ли адаптировать этот метод для RISC-компьютеров, на которых нет команды *перехода при наличии переноса и ненулевого результата команды*. Примем для простоты размер блока равным 8. Согласно [7] переход по метке CROSSES произойдет только в том случае, если был перенос (т.е. $(a \mid -8) + l > 2^{32}$) и если результат не равен нулю (т.е. $(a \setminus -8) + / * 2^{32}$), что эквивалентно условию $(a \setminus -8) + l > 2^{32}$. Это условие, в свою очередь, эквивалентно появлению переноса в последнем сложении при вычислении $((a \mid -8) - 1) + l$. Если в компьютере имеется команда *переход при наличии переноса,* то данный алгоритм можно использовать непосредственно; при этом потребуется пять команд с учетом загрузки константы -8.

Если же такой команды нет, можно воспользоваться тем, что перенос при суммировании $x + y$ происходит тогда и только тогда, когда $\neg x \overset{u}{<} y$, и будет получено выражение

$$\neg((a \mid -8) - 1) \overset{u}{<} l .$$

Использование различных тождеств типа $\neg(x - 1) = -x$ дает ряд эквивалентных выражения для предиката "выход за границы блока".

$$-(a \mid -8) \overset{u}{<} l$$
$$\neg(a \mid -8) + 1 \overset{u}{<} l$$
$$(\neg a \,\&\, 7) + 1 \overset{u}{<} l$$

На большинстве RISC-компьютеров эти выражения вычисляются за пять или шесть команд.

С другой стороны, очевидно, что выход за границы 8-байтового блока произойдет тогда и только тогда, когда $(a \,\&\, 7) + l - 1 \geq 8$. Непосредственному вычислению это выражение не поддается в силу возможного переполнения (которое возникает при очень больших l). Однако если переписать выражение в виде $8 - (a \,\&\, 7) < l$, то его можно вычислить, не вызывая переполнения.

Это дает нам выражение

$$8 - (a \,\&\, 7) \overset{u}{<} l ,$$

которое на большинстве RISC-компьютеров вычисляется при помощи пяти команд (или четырех, если компьютер имеет команду *вычитания из непосредственно заданного значения).* При наличии выхода за границы блока одна дополнительная команда вычитания позволяет вычислить значение $l - (8 - (a \,\&\, 7))$, которое дает нам длину поддиапазона адресов, выходящего за пределы первого блока.

ГЛАВА 4

АРИФМЕТИЧЕСКИЕ ГРАНИЦЫ

4.1. Проверка границ целых чисел

Под "проверкой границ" *(bounds checking)* подразумевается проверка того, находится ли целое значение x в пределах диапазона от a до b, т.е. выполняется ли соотношение

$$a \le x \le b .$$

Считаем, что все упомянутые здесь величины представляют собой целые знаковые числа.

Важным применением данного действия является проверка индексов элементов массива. Например, пусть объявлен одномерный массив *A,* нижняя и верхняя граница которого соответственно равны 1 и 10. При обращении к элементу массива $A(i)$ компилятор может сгенерировать код для проверки корректности индекса $1 \le i < 10$. Если i выходит за пределы объявленного диапазона индексов, выполняется команда *перехода* или *прерывания* для обработки ошибки. В этом разделе показано, что такая проверка может быть выполнена посредством одного сравнения [52]:

$$i - 1 \overset{u}{\le} 9 .$$

Очевидно, что этот метод лучше предыдущего, поскольку содержит только одну команду *условного ветвления;* кроме того, значение $i - 1$ в дальнейшем можно использовать при адресации элемента массива.

Возникает вопрос: всегда ли справедлива реализация

$$a \le x \le b \Rightarrow x - a \overset{u}{\le} b - a$$

(включая случай переполнения при вычитании)? Да, при выполнении условия $a \le b$. В случае проверки границ массива правила языка программирования могут потребовать, чтобы массив не мог иметь нулевого или отрицательного числа элементов, причем проверка выполнения этого правила может выполняться во время компиляции программы или при выделении памяти для динамических массивов. В этом случае рассмотренное выше преобразование вполне корректно.

Для начала докажем следующую лемму.

Лемма. *Если a и b — знаковые целые числа, причем* $a < b$*, и если рассматривать вычисленное значение* $b - a$ *как беззнаковое целое число, то это значение корректно представляет арифметическое значение* $b - a$.

Доказательство. (Предполагается, что используется 32-разрядный компьютер.) Так как $a \le b$, то значения арифметической разности Ъ-а находятся в диапазоне от 0 до $(2^{31} - 1) - (-2^{31}) = 2^{32} - 1$. Если истинное значение разности находится в диапазоне от 0 до $2^{31} - 1$, то вычисленная компьютером разность будет корректна (так как бит знака равен О при рассмотрении результата как целого знакового числа). Таким образом, вычисленный результат будет корректен независимо от того, как он интерпретируется — как знаковое или беззнаковое целое число.

Если значение истинной разности находится между 2^{31} и $2^{32}-1$, вычисленный компьютером результат будет отличаться от истинного на 2^{32} (поскольку это число не может быть представлено 32 битами при знаковой интерпретации). Таким образом, при знаковой интерпретации будет получен результат, лежащий в диапазоне от -2^{32} до -1. Результат машинного вычисления также уменьшается на 2^{32}, при этом бит знака становится равным 1. Однако если рассматривать вычисленный результат как беззнаковое целое число, то его значение за счет использования бита знака (который при этом имеет вес 2^{31}) увеличивается на 2^{32}. Следовательно, при рассмотрении результата как беззнакового целого числа он корректно представляет истинное значение.

Теперь мы можем перейти к "теореме границ".

Теорема. *Если аи b — целые знаковые числа и $a < b$, то*

$$a \le x \le b = x - a \overset{u}{\le} b - a \,. \qquad (1)$$

Доказательство. В зависимости от значения x, возможны три случая. Согласно рассмотренной ранее лемме, поскольку в любом из случаев $a \le b$, значение $b - a$, рассматриваемое как беззнаковое целое число (что и происходит в уравнении (1)), равно арифметическому значению разности b и a.

Случай 1: $x < a$. При этом разность $x - a$, рассматриваемая как беззнаковое число, равна $x\text{-}a + 2^{32}$. Какими бы ни были значения x и b (в диапазоне 32-битовых чисел), выполняется неравенство $x + 2^{32} > b$ и соответственно $x - a + 2^{32} > b - a$. Следовательно, $x - a \overset{u}{>} b - a$ и обе части уравнения (1) ложны.

Случай 2: $a < x < b$. В этом случае арифметически $x - a < b - a$. Так как $a < x$, то согласно лемме истинное значение разности $x - a$ равно вычисленному значению $x - a$, если рассматривать его как беззнаковое. Следовательно, $x - a \overset{u}{\le} b - a$ и обе части уравнения (1) истинны.

Случай 3: $x > b$. В таком случае $x - a > b - a$. Так как $b > a$, в этом случае $x > a$, и согласно лемме истинное значение разности $x - a$ равно вычисленному значению $x - a$, рассматриваемому как беззнаковое число. Следовательно, $x - a \overset{u}{>} b - a$ и обе части уравнения (1) ложны.

Рассмотренная теорема справедлива и для *беззнаковых* целых a и b, Для беззнаковых чисел рассмотренная лемма становится тривиальной, а значит, приведенное выше доказательство теоремы справедливо и для беззнаковых чисел.

Ниже приведено несколько аналогичных преобразований, полученных из теоремы границ и применимых как для знаковых, так и для беззнаковых целых a, b их.

$$
\begin{aligned}
&\text{Если } a < b, \text{ то } && a \le x \le b = x - a \overset{u}{\le} b - a = b - x \overset{u}{\le} b - a. \\
&\text{Если } a \le b, \text{ то } && a \le x < b = x - a \overset{u}{<} b - a. \\
&\text{Если } a \le b, \text{ то } && a < x \le b = b - x \overset{u}{<} b - a. \\
&\text{Если } a < b, \text{ то } && a < x < b = x - a - 1 \overset{u}{<} b - a - 1 = b - x - 1 \overset{u}{<} b - a - 1.
\end{aligned}
\qquad (2)
$$

В последнем преобразовании $b - a - 1$ можно заменить на $b + \neg a$.

При проверке граничных условий вида $-2^{n-1} \le x \le 2^{n-1} - 1$ лучше использовать другие преобразования. Проверка таких условий определяет, может ли знаковое число x быть корректно представлено как n-разрядное целое число в дополнительном коде. Ниже приводится несколько равноценных методов проверки для $n = 8$.

а) $-128 \le x \le 127$

б) $x + 128 \overset{u}{\le} 255$

в) $\left(x \overset{s}{\gg} 7\right) + 1 \overset{u}{\le} 1$

г) $x \overset{s}{\gg} 7 = x \overset{s}{\gg} 31$

д) $\left(x \overset{s}{\gg} 7\right) + \left(x \overset{u}{\gg} 31\right) = 0$

е) $\left(x \ll 24\right) \overset{s}{\gg} 24 = x$

ж) $x \oplus \left(x \overset{s}{\gg} 31\right) \le 127$

Соотношения (б) и (в) получены непосредственно из предыдущего материала главы (в формуле (в) преобразование применено к значению x, сдвинутому вправо на 7 бит). Выражения (в)-(е) (а, возможно, и (ж)), вероятно, следует использовать, только если константы из (а) и (б) превышают размеры полей непосредственно задаваемых значений в командах *сравнения* и *сложения*.

Для еще одного частного случая проверки границ применимо преобразование

$$0 \le x \le 2^n - 1 \Leftrightarrow \left(x \overset{u}{\gg} n\right) = 0$$

или в более общем виде

$$a \le x \le a + 2^n - 1 \Leftrightarrow \left((x - a) \overset{u}{\gg} n\right) = 0 .$$

4.2. Определение границ суммы и разности

Ряд оптимизирующих компиляторов в процессе работы выполняют анализ области значений выражений, который представляет собой процесс определения верхней и нижней границ значений арифметического выражения в программе. Хотя такая оптимизация реально не приводит к большому выигрышу, тем не менее она позволяет вносить в программу определенные улучшения, например избежать проверки диапазона в операторе выбора switch языка С или проверки значений индексов в режиме отладки.

Предположим, что диапазон значений двух переменных x и y задается следующим образом (все величины беззнаковые):

$$\begin{aligned} a \le x \le b, \\ c \le y \le d. \end{aligned} \tag{3}$$

Каковы в этом случае компактные границы значений выражений $x + y$, $x - y$ и $-x$? Очевидно, что арифметически $a + c \le x + y \le b + d$. Но проблема в том, что при вычислении вполне может возникнуть переполнение.

Один из способов определения диапазона значений основан на следующей теореме.

Теорема. *Если a, b, c, d, x и y — беззнаковые целые числа, причем*

$$a \overset{u}{\le} x \overset{u}{\le} b \quad \text{и}$$
$$c \overset{u}{\le} y \overset{u}{\le} d,$$

то

$$\begin{array}{ll} 0 \overset{u}{\le} x + y \overset{u}{\le} 2^{32} - 1, & \text{если } a + c \le 2^{32} - 1 \text{ и } b + d \ge 2^{32}, \\ a + c \overset{u}{\le} x + y \overset{u}{\le} b + d & \text{в противном случае;} \end{array} \qquad (4)$$

$$\begin{array}{ll} 0 \overset{u}{\le} x - y \overset{u}{\le} 2^{32} - 1, & \text{если } a - d < 0 \text{ и } b - c \ge 0, \\ a - d \overset{u}{\le} x - y \overset{u}{\le} b - c & \text{в противном случае;} \end{array} \qquad (5)$$

$$\begin{array}{ll} 0 \overset{u}{\le} -x \overset{u}{\le} 2^{32} - 1, & \text{если } a = 0 \text{ и } b \ne 0, \\ -b \overset{u}{\le} -x \overset{u}{\le} -a & \text{в противном случае.} \end{array} \qquad (о)$$

Неравенство (4) гласит, что значения суммы $x + y$ *обычно* находятся в диапазоне от $a + c$ до $* + d$. Однако если при вычислении $b + d$ было переполнение, а при вычислении $a + c$ переполнения *не* было, то нижней и верхней границами суммы $x + y$ будут соответственно 0 и максимальное беззнаковое целое число. Выражения (5) интерпретируются аналогично, но значение истинной разности, меньшее 0, означает, что имело место переполнение (в отрицательном направлении).

Доказательство. Пусть значения x и y находятся в указанных выше диапазонах. Если при вычислении обеих сумм — $a + c$ и $b + d$ — переполнения не было, то и при вычислении суммы $x + y$ переполнения не будет и вычисленный результат будет правильным (что и утверждает второе неравенство из выражения (4)). Если же переполнение возникло при вычислении как $a + c$, так и $b + d$, то при вычислении $x + y$ также произойдет переполнение. С арифметической точки зрения ясно, что

$$a + c - 2^{32} \le x + y - 2^{32} \le b + d - 2^{32}.$$

Но именно так и выполняются вычисления в случае, когда переполнение происходит во всех трех суммах. Следовательно, и в этом случае

$$a + c \overset{u}{\le} x + y \overset{u}{\le} b + d.$$

Если при вычислении $a + c$ переполнения нет, а при вычислении $b + d$ — есть, то

$$a + c \le 2^{32} - 1 \text{ и } b + d \ge 2^{32}.$$

Поскольку $x + y$ может принимать все значения из диапазона от $a + c$ до $b + d$, значение этой суммы находится между $2^{32} - 1$ и 2^{32}, т.е. вычисленное значение суммы $x + y$ будет находиться между $2^{32} - 1$ и 0 (хотя и не принимает *все* значения из указанного диапазона).

Случай, когда при вычислении $a + c$ возникает переполнение, а при вычислении $b + d$ не возникает, невозможен, поскольку $a \le b$ и $c \le d$.

Доказательство неравенств (4) завершено. Неравенства (5) доказываются аналогично, с тем отличием, что здесь "переполнение" означает, что истинное значение разности меньше 0.

При доказательстве неравенств (6) можно воспользоваться неравенствами (5), в которых $a = b = 0$, а затем переименовать переменные. (Выражение $-x$, где x — беззнаковое число, означает вычисление значения $2^{32} - x$ или $\neg x + 1$ — что вам больше нравится.)

Поскольку беззнаковое переполнение легко распознается (см. раздел 2.12), полученные результаты несложно перевести в код, показанный в листинге 4.1, где вычисляется верхняя (переменная s) и нижняя (переменная t) границы суммы и разности.

Листинг 4.1. Верхняя и нижняя границы беззнаковых суммы и разности

```
S = a + c;
t = b + d;
if (s >= a && t < b)
{
    S = 0;
    t = 0xFFFFFFFF;
}
```

```
s = a - d;
t = b - C;
if (s > a && t <= b)
{
    s = 0;
    t = 0xFFFFFFFF;
}
```

Знаковые числа

При сложении и вычитании целых знаковых чисел не все так просто. Как и прежде, предположим, что диапазоны значений x и y задаются следующим образом (все величины здесь — знаковые):

$$\begin{aligned} a < x < Ь, \\ c < y < d. \end{aligned} \tag{7}$$

Необходимо определить компактные границы значений $x+y$, $x-y$ и $-ж$. Доказательство формул для знаковых величин очень похоже на доказательство в беззнаковом случае, так что приведем только конечный результат для сложения.

$$\begin{aligned}
a+c<-2^{31}, b+d<-2^{31} &: a+c \le x+y \le b+d \\
a+c<-2^{31}, b+d \ge -2^{31} &: -2^{31} < x+y \le 2^{31}-1 \\
-2^{31} \le a+c<2^{31}, b+d<2^{31} &: a+c < x+y \le b+d \\
-2^{31} \le a+c<2^{31}, b+d \ge 2^{31} &: -2^{M} < x+y \le 2^{31}-1 \\
a+c \ge 2^{31}, b+d \ge 2^{31} &: a+c \le x+y \le b+d
\end{aligned} \tag{8}$$

Первая строка означает, что если при вычислении обеих сумм — $a+c$ и $b+d$ — было переполнение в отрицательном направлении, то вычисленная сумма $x+y$ будет находиться между вычисленными суммами $a+c$ и $b+d$, так как все три вычисленные суммы оказываются слишком велики на одну и ту же величину (2^{32}). Вторая строка означает, что если при вычислении суммы $a+c$ происходит переполнение в отрицательном направлении, а при вычислении $b+d$ переполнения либо не было, либо оно было в положительном направлении, то вычисленное значение суммы $x+y$ находится между крайним отрицательным и крайним положительным числом (хотя и может принимать не все значения из указанного диапазона). Остальные строки интерпретируются аналогично.

Если представить выражение для границ y в виде $-d < -y < -c$ и воспользоваться уже имеющимися выражениями для суммы, то получим формулы, по которым определяется диапазон значений в результате вычитания знаковых чисел.

$$\begin{aligned}
a-d<-2^{31}, b-c<-2^{31} &: a-d \le x-y \le b-c \\
a-d<-2^{31}, b-c \ge -2^{31} &: -2^{31} < x-y \le 2^{31}-1 \\
-2^{31} \le a-d<2^{31}, b-c<2^{31} &: a-d \le x-y \le b-c \\
-2^{31} \le a-d<2^{31}, b-c \ge 2^{31} &: -2^{31} \le x-y \le 2^{31}-1 \\
a-d \ge 2^{31}, b-c \ge 2^{31} &: a-d \le x-y \le b-c
\end{aligned}$$

Формулы для отрицания можно получить из выражений для разности знаковых чисел в предположении $a = b = 0$, отбросив ряд невозможных комбинаций, переименовав переменные и проведя некоторые упрощения.

$$a = -2^{31}, b = -2^{31}: \ -x = -2^{31}$$
$$a = -2^{31}, b \neq -2^{31}: \ -2^{31} < -x \leq 2^{31} - 1$$
$$a \neq -2^{31}: \ -b \leq x \leq -a$$

Программа на языке С, вычисляющая границы арифметических выражений, несколько сложновата, поэтому рассмотрим только код определения диапазона значений суммы двух знаковых операндов. По-видимому, проще всего выполнить проверку границ в двух случаях из (8) — когда нижняя и верхняя границы равны крайним отрицательному и положительному числам соответственно. Переполнение в отрицательном направлении имеет место, если оба операнда отрицательны, а их сумма — неотрицательна (см. раздел 2.12). Таким образом, для проверки условия $a + c < -2^{31}$ сначала необходимо вычислить сумму `s = a + c;`, а затем использовать код наподобие `if (a<o && c<o && s>=0) ...`. Однако программа будет более эффективной[1], если все логические операции будут выполняться непосредственно над арифметическими переменными, при этом результат будет находиться в бите знака. Тогда рассмотренное условие можно представить иначе: `if ((a & c & -s) < o) ...`. Таким образом, получаем фрагмент кода, приведенный в листинге 4.2.

Листинг 4.2. Определение границ знакового сложения

```
s = a + c;
t = b + d;
u = a & c & ~s & ~(b & d & ~t);
v = ((a ^ c) | ~(a ^ s» & (~b & -d & t) ;
if ((u | v) < 0}
{
    S = 0x80000000;
    t = OX7FFFFFFF;
}
```

Переменная u имеет значение true (т.е. бит знака равен 1), если при вычислении a + c происходит переполнение в отрицательном направлении, а при вычислении b + d переполнения в отрицательном направлении *нет*. Переменная v имеет значение true, если при вычислении a + c переполнения *нет*, а при вычислении b + d имеет место переполнение в положительном направлении. Первое условие можно записать как "a и c имеют разные знаки, или a и s имеют один и тот же знак". Проверка `if((u|v)<0)` эквивалентна проверке `if(u<0||v<0)`, т.е. если истинно или u, или v (или оба одновременно).

4.3. Определение границ логических выражений

Как и в предыдущем разделе, предположим, что границы двух переменных x и y задаются следующим образом (все величины беззнаковые):

$$\begin{aligned} a \leq x \leq b, \\ c \leq y \leq d. \end{aligned} \tag{9}$$

---

[1] Другими словами, более компактной, с минимальным количеством переходов. В предположении, что границы обычно не слишком велики, проверку следует начинать для случая отсутствия переполнения, так как велика вероятность получить результат сразу же, в ходе этой проверки.

Необходимо определить компактные границы выражений $x \mid y$, $x \& y$, $x \oplus y$ и $\neg x$.

Комбинируя неравенства (9) с неравенствами из раздела 2.3 и учитывая, что $\neg x = 2^м - 1 - x$, получим

$$\begin{aligned} \max(a, c) \le (x \mid y) &\le b + d, \\ 0 \le (x + y) &\le \min(b, d), \\ 0 \le (x \oplus y) &\le b + d, \\ \neg b \le \neg x &\le \neg a, \end{aligned}$$

где предполагается, что при сложении $b + d$ переполнения не возникает. Эти формулы легко вычисляются и достаточно эффективны для использования в компиляторах, о чем упоминалось в предыдущем разделе. Однако границы в первых двух неравенствах не являются компактными. Пусть, например, диапазон значений переменных *x ну в* двоичной записи задан следующим образом:

$$\begin{aligned} 00010 < x < 00100, \\ 01001 < y < 10100. \end{aligned} \tag{10}$$

После проверки (перебора всех 36 возможных комбинаций значений *x* и *y)* получим, что $01010 \le (x \mid y) < 10111$, т.е. нижняя граница не равна ни $\max(a, c)$, ни $a \backslash c$; верхняя граница также не равна ни $b + d$, *ни* $b \backslash d$.

Можно ли определить компактные границы логических выражений, если известны значения *а, Ь, с* и *d* из (9)? Давайте сначала найдем минимальное значение выражения $x \backslash y$. Естественно предположить, что значение этого выражения будет минимальным при минимальных значениях *x* **и y**, равным $a \backslash c$. Однако, как видно из примера (10), минимальное значение на самом деле может оказаться меньше предполагаемого.

Для поиска минимума начнем со значений $x = a$ и $y = c$ и будем искать значения дс и *у,* при которых значение выражения $x \backslash y$ будет минимальным. Этот результат и будет искомым минимальным значением. Вместо того чтобы присваивать значения ***а* и *с*** переменным *x* и *у,* будем работать непосредственно со значениями а и с, увеличивая одно из них так, чтобы снижать значение $a \backslash c$.

Процедура состоит в сканировании битов а и с слева направо. Если оба бита равны 0, то в соответствующем разряде результата будет 0. Если оба бита равны 1, то и соответствующий бит результата будет равен 1 (очевидно, никакие другие значения *x* **и** *у* не могут уменьшить полученный результат). В обоих этих случаях переходим к сравнению следующих разрядов. Если в следующем разряде одной из границ содержится единичный бит, а у другой границы — нулевой, то, возможно, замена 0 на 1 в этом разряде и установка всех следующих за ним битов в 0 снизит значение $a \mid c$. Такое изменение границ не может увеличить значение $a \backslash c$, так как в любом случае в этом разряде результата будет 1 за счет другой границы. Итак, мы формируем число с нулевым битом, замененным на единичный, а все последующие биты этого числа заменяем нулями. Если полученное таким образом число меньше или равно соответствующей верхней границе, такая замена возможна. Выполним эту замену, а затем применим операцию *или* к полученному значению и другой нижней границе. Если же после описанной замены битов новое значение нижней границы оказалось больше верхней, то такая замена невозможна, и мы переходим к следующим битам.

Вот и все. Казалось бы, после замены следует продолжить сканирование, чтобы искать дальнейшие возможности улучшения полученного результата — еще меньшее зна-

чение $a \mid c$. Однако, даже если в одном из следующих разрядов можно заменить 0 на 1, установка следующих за ним битов равными нулю не даст меньшего значения $a \mid c$, так как в этих разрядах уже содержатся нули.

Программа на языке С для данного алгоритма представлена в листинге 4.3. Предполагается, что оптимизирующий компилятор вынесет вычисление подвыражений `~a&c` и `a&~c` за пределы цикла. Если у нас есть функция вычисления *количества ведущих нулевых битов,* код можно ускорить, инициализируя m следующим образом:

```
m = 0x80000000 >> nlz(a^c);
```

При этом в a ^c опускаются те разряды, в которых изначально у обеих констант оба бита равны 0 или оба равны 1. Это ускорение работы программы будет особенно эффективным при `a^c` равным 0 (т.е. если a = c); при этом сдвиг вправо должен выполняться по модулю 64. Если функции nlz в вашем распоряжении нет, можно воспользоваться одной из версий функции flp2 (см. раздел 3.2) с аргументом a ^ c.

Листинг 4.3. Вычисление нижней границы $x \mid y$

```
unsigned minOR(unsigned a, unsigned b,
               unsigned c, unsigned d)
{
    unsigned m, temp;

    m = 0X80000000;
    while (m != 0)
    {
        if (~a & c & m)
        {
            temp = (a | m) & -m;
            if (temp <= b) {a = temp; break;}
        }
        else if (a & -c & m)
        {
            temp = (c | m) & -m;
            if (temp <= d) {c = temp; break;}
        }
        m = m >> 1;
    }
    return a | c;
}
```

Рассмотрим теперь алгоритм поиска *максимального* значения выражения $x \mid y$, где, как и ранее, значения переменных ограничены неравенствами (9). Этот алгоритм аналогичен алгоритму поиска минимального значения. Верхние границы поразрядно сканируются слева направо до тех пор, пока в некоторой позиции биты в b и rf не окажутся равными единицам. Если такая позиция обнаружена, алгоритм пытается увеличить значение $b \mid d$, уменьшая одну из границ заменой в соответствующем разряде 1 на 0 и заменяя все последующие биты единичными. Если такая замена корректна (т.е. полученное таким образом новое значение верхней границы не ниже соответствующей нижней границы), то изменения принимаются. Если же замена оказывается некорректной, пытаемся изменить другую границу. Если и эта замена некорректна, сканируются следующие биты. Программа на языке С, реали-

зующая данный алгоритм, приведена в листинге 4.4. Здесь за пределы цикла можно вынести вычисление подвыражения b & d, а кроме того, работу программы можно ускорить следующей инициализацией:

```
m = 0x80000000 >> nlz(b & d) ;
```

Листинг 4.4. Вычисление верхней границы $x \mid y$

```
unsigned maxOR (unsigned a, unsigned b,
                unsigned c, unsigned d)
{
    unsigned m,  temp;

    m = 0x80000000;
    while (m != 0)
    {
        if (b & d & m)
        {
            temp = (b - m)    (m - 1) ;
            if (temp >= a) {b = temp; break;}
            temp = (d - m)    (m - 1) ;
            if (temp >= c) {d = temp; break;}
        }
        m = m >> 1;
    }
    return b | d;
}
```

Определить диапазон значений выражения $x \& y$, где границы x и y определены неравенствами (9), можно с помощью двух методов: алгебраического и непосредственного вычисления. В основе алгебраического метода лежит закон де Моргана (DeMorgan):

$$x \& y = \neg(\neg x \mid \neg y).$$

Поскольку мы знаем, как определить точные границы выражения $x \mid y$, границы $x \& y$ вычисляются тривиально с помощью команды *не* ($a \overset{u}{\le} x \overset{u}{\le} b \Leftrightarrow \neg b \overset{u}{\le} \neg x \overset{u}{\le} \neg a$):

$$\text{minAND}(a,b,c,d) = \neg\text{maxOR}(\neg b, \neg a, \neg d, \neg c),$$
$$\text{maxAND}(a,b,c,d) = \neg\text{minOR}(\neg b, \neg a, \neg d, \neg c).$$

Код вычисления этих функций, приведенный в листингах 4.5 и 4.6, очень похож на код вычисления границ выражения с *или*.

Листинг 4.5. Вычисление нижней границы $x \& y$

```
unsigned minAND (unsigned a, unsigned b,
                 unsigned c, unsigned d)
{
    unsigned m, temp;

    m = 0x80000000;
    while (m != 0)
    {
        if (~a & -c & m)
        {
            temp = (a | m) & -m;
            if (temp <= b) {a = temp; break;}
```

```
            temp = (c | m) & -m;
            if (temp <= d) {c = temp; break;}
        }
        m = m >> 1;
    }
    return a & c;
}
```

Листинг 4.6. Вычисление верхней границы $x \,\&\, y$

```
unsigned maxAND(unsigned a, unsigned b,
                unsigned c, unsigned d)
{
    unsigned m, temp;

    m = 0x80000000;
    while (m != 0)
    {
        if (b & -d & m)
        {
            temp = (b & ~m) | (m - 1) ;
            if (temp >= a) {b = temp; break;}
        }
        else if (~b & d & m)
        {
            temp = (d & ~m) | (m - 1) ;
            if (temp >= c) {d = temp; break;}
        }
        m = m >> 1;
    }
    return b & d;
}
```

Алгебраический метод поиска границ можно использовать для любых побитовых логических выражений, кроме *исключающего или* и *эквивалентности*. Причина в том, что при выражении этих операций через команды *и, или* и *не* получаются два члена, содержащие x ч y. Например, для *исключающего или* имеем

$$\min_{\substack{a \le x \le b \\ c \le y \le d}} (x \oplus y) = \min_{\substack{a \le x \le b \\ c \le y \le d}} \left((x \,\&\, \neg y) \,|\, (\neg x \,\&\, y) \right).$$

Указанные операнды команды *или* нельзя минимизировать по отдельности (не доказав предварительно, что так можно делать), поскольку мы ищем одно значение x и одно значение y, которые минимизируют выражение *или* в целом.

Для вычисления границ выражения с исключающим *или* можно использовать следующие выражения:

$$\text{minXOR}(a,b,c,d) = \text{minAND}(a,b,\neg d,\neg c) \,|\, \text{minAND}(\neg b,\neg a,c,d),$$
$$\text{maxXOR}(a,b,c,d) = \text{maxOR}(0, \text{maxAND}(a,b,\neg d,\neg c), 0, \text{maxAND}(\neg b,\neg a,c,d)).$$

Значения minXOR и maxOR несложно вычислить непосредственно. Код для вычисления minXOR такой же, как и для вычисления minOR (см. листинг 4.3), нужно лишь удалить два оператора break, а в последней строке заменить возвращаемое значение на a ^ c. Код для вычисления maxXOR тот же, что и для вычисления maxOR (см. листинг 4.4), за исключением того, что четыре строки после оператора if необходимо заменить фрагментом

```
temp = (b - m) | (m - 1) ;
if (temp >= a) b = temp;
else
{
    temp = (d - m) | (m - 1) ;
    if (temp >= c) d = temp;
}
```

Кроме того, возвращаемое значение нужно заменить на b ^ d.

Знаковые границы

Вычисление целых *знаковых* границ для логических выражений значительно усложняется. Вычисления границ зависят от расположения числа 0 по отношению к заданным диапазонам (от a до b и от c до d), и для каждого частного случая используются свои формулы. В табл. 4.1 приведен один из методов вычисления верхней и нижней границ выражения $x|y$. Если значение переменной больше или равно 0, в соответствующей графе таблицы стоит знак "плюс", если значение переменной меньше 0 — знак "минус". Под заголовком minOR приведены выражения, вычисляющие нижнюю границу $x \setminus y$ для данных диапазонов, а под заголовком maxOR — формулы для вычисления верхней границы. Вычислить верхнюю и нижнюю границы выражения $x \setminus y$ можно следующим образом: из битов знаков чисел a, b, c и d составляется 4-битовое число, значение которого находится в пределах от 0 до 15, и используется оператор выбора switch. Обратите внимание, что будут задействованы не все значения от 0 до 15, так как невозможны ситуации, когда $a > b$ или $c > d$.

Для знаковых чисел справедливо отношение

$$a \le x \le b \Leftrightarrow \neg b \le \neg x \le \neg a,$$

так что для распространения результатов из табл. 4.1 на другие логические выражения (кроме *исключающего или* и *эквивалентности)* может использоваться алгебраический метод. Выражения, определяющие границы других логических команд, читателю предлагается получить самостоятельно.

Таблица 4.1. Вычисление знаковых функций minOR и maxOR с помощью беззнаковых

a	b	c	d	minOR(знаковый)	maxOR (знаковый)
-	-	-	-	$\text{minOR}(a,b,c,d)$	$\text{maxOR}(a,b,c,d)$
-	-	-	+	a	-1
-	-	+	+	$\text{minOR}(a,b,c,d)$	$\text{maxOR}(a,b,c,d)$
-	+	-	-	c	-1
-	+	-	+	$\min(a,c)$	$\text{maxOR}(0,b,0,d)$
-	+	+	+	$\text{minOR}(a,\text{0xFFFFFFFF},c,d)$	$\text{maxOR}(0,b,c,d)$
+	+	-	-	$\text{minOR}(a,b,c,d)$	$\text{maxOR}(a,b,c,d)$
+	+	-	+	$\text{minOR}(a,b,c,\text{0xFFFFFFFF})$	$\text{maxOR}(a,b,0,d)$
+	+	+	+	$\text{minOR}(a,b,c,d)$	$\text{maxOR}(a,b,c,d)$

ГЛАВА 5

ПОДСЧЕТ БИТОВ

5.1. Подсчет единичных битов

На компьютере IBM Stretch (выпущенном в 1960 году) имелась возможность подсчета количества единичных битов в слове и количества ведущих нулевых битов слова (причем эти величины являлись побочным результатом выполнения любой логической операции!). Функция подсчета количества единичных битов иногда носила название степени заполнения (population count), например на машинах Stretch или SPARCv9.

Если на машине нет отдельной команды для вычисления этой функции, то для подсчета единичных битов в слове можно воспользоваться другими методами. Обычно исходное слово делится на двухразрядные поля и в каждое поле помещается количество имевшихся в нем единичных битов. Затем значения, содержащиеся в соседних двухразрядных полях, складываются и результат помещается в четырехразрядное поле и т.д. Более подробно этот метод обсуждается в [54]. Этот метод проиллюстрирован на рис. 5.1. Слово, в котором подсчитывается количество единичных битов, находится в первой строке, в последней строке содержится результат (равный 23 в десятичной системе счисления).

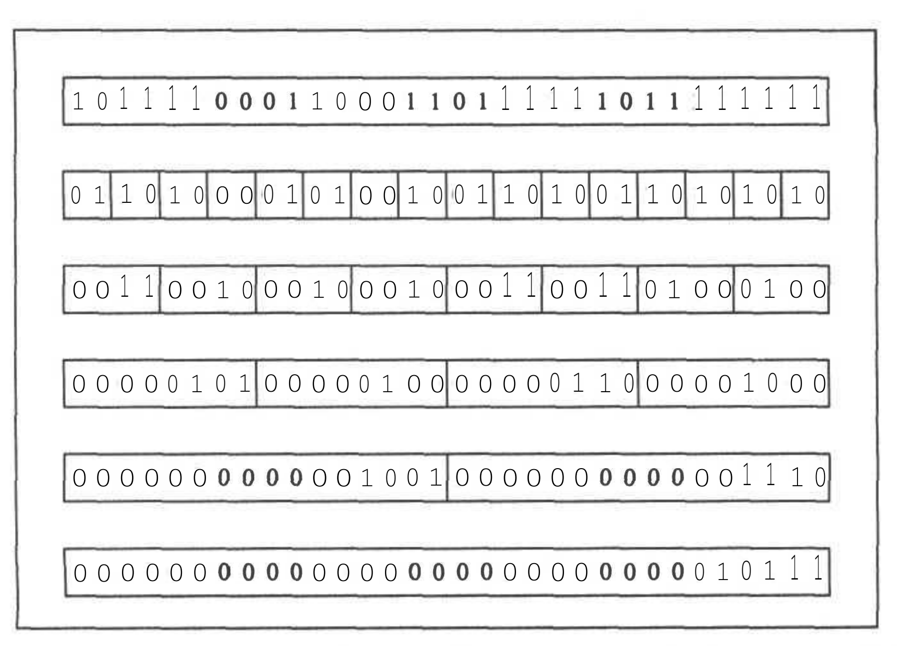


Рис. 5.1. Подсчет количества единичных битов, стратегия "разделяй и властвуй"

Рассмотренный метод — пример стратегии "разделяй и властвуй": исходная задача (суммирование в 32-битовом слове) разбивается на две подзадачи (суммирование в 16-битовом слове), каждая из которых решается независимо от другой. Затем результаты объединяются (в нашем случае суммируются). Данная стратегия является рекурсивной: 16-битовые слова обрабатываются как два 8-битовых и т.д.

Выполнение подзадач на самом нижнем уровне (суммирование значений соседних битов) можно осуществлять параллельно, как и суммирование значений в соседних полях, которое можно выполнить параллельно за фиксированное число шагов на любом этапе. Весь алгоритм может быть выполнен за $\log_2(32) = 5$ шагов.

Другими примерами стратегии "разделяй и властвуй" являются хорошо всем известные алгоритмы бинарного поиска, быстрой сортировки, а также метод реверса битов слова, рассматриваемый в разделе 7.1.

Метод, приведенный на рис. 5.1, может быть выражен на языке С следующим образом:

```
x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF);
```

В первой строке можно было бы написать, пожалуй, более естественное выражение (x & 0xAAAAAAAA) >> 1, но в данном случае лучше использовать приведенное выражение (x >> 1) & 0x55555555, так как при этом удается избежать загрузки в регистры двух больших констант, что на машине с отсутствующей командой *и-не* будет стоить дополнительной команды. Аналогичные замечания справедливы и для остальных строк кода.

Очевидно, что последняя команда *и* не является необходимой; кроме того, если точно известно, что при суммировании полей не произойдет переноса в соседнее поле, остальные команды *и* тоже не нужны. Этот код можно еще немного усовершенствовать, а именно преобразовать первую строку так, чтобы в ней выполнялось на одну команду меньше. Такой упрощенный код показан в листинге 5.1; он не содержит условных переходов и выполняется за 21 команду.

Листинг 5.1. Подсчет количества единичных битов в слове

```
int pop(unsigned x)
{
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    x = x + (x >> 8);
    x = x + (x >> 16);
    return x & 0x0000003F;
}
```

Первое присвоение x основано на первых двух членах одной удивительной формулы:

$$\mathrm{pop}(x) = x - \left\lfloor \frac{x}{2} \right\rfloor - \left\lfloor \frac{x}{4} \right\rfloor - \ldots - \left\lfloor \frac{x}{2^{31}} \right\rfloor. \qquad (1)$$

В формуле (1) значение x должно быть не меньше 0. Если x — целое беззнаковое число, формула (1) может быть реализована в виде последовательности из 31 команды *сдвига вправо на единицу* и 31 команды *вычитания*. Процедура в листинге 5.1 использует параллельно первые два члена этой формулы для каждого двухбитового поля.

Существует простое доказательство уравнения (1), приведенное ниже для случая 4-битового слова. Пусть имеется слово $b_3b_2b_1b_0$, где каждое b_i равно 0 или 1. Тогда

$$x-\left\lfloor\frac{x}{2}\right\rfloor-\left\lfloor\frac{x}{4}\right\rfloor-\left\lfloor\frac{x}{8}\right\rfloor=b_3\cdot 2^3+b_2\cdot 2^2+b_1\cdot 2^1+b_0\cdot 2^0-$$
$$-\left(b_3\cdot 2^2+b_2\cdot 2^1+b_1\cdot 2^0\right)-$$
$$-\left(b_3\cdot 2^1+b_2\cdot 2^0\right)-$$
$$-\left(b_3\cdot 2^0\right)=$$
$$=b_3\left(2^3-2^2-2^1-2^0\right)+b_2\left(2^2-2^1-2^0\right)+b_1\left(2^1-2^0\right)+b_0\left(2^0\right)$$
$$=b_3+b_2+b_1+b_0.$$

Формулу (1) можно получить и иначе: например, заметив, что i-й бит в двоичном представлении неотрицательного числа x вычисляется по формуле

$$b_i=\left\lfloor\frac{x}{2^i}\right\rfloor-2\left\lfloor\frac{x}{2^{i+1}}\right\rfloor,$$

и вычисляя сумму всех b_i, где i принимает значения от 0 до 31. Последний член будет равен нулю, так как $x<2^{32}$.

Формулу (1) можно обобщить для других систем счисления. Например, для системы счисления с основанием 10 получим следующее выражение:

$$\text{sum_digits}(x)=x-9\left\lfloor\frac{x}{10}\right\rfloor-9\left\lfloor\frac{x}{100}\right\rfloor-\ldots,$$

где слагаемые вычисляются до тех пор, пока их значение не становится равным нулю. Полученная формула доказывается аналогично предыдущей.

Если применить описанный алгоритм для системы счисления по основанию 4, можно получить другой код для вычисления функции $\text{pop}(x)$, который аналогичен приведенному в листинге 5.1, но вторая строка которого заменяется строкой

```
x = x - 3 * ((x >> 2) & 0x33333333);
```

Этот код выполняется за те же шесть команд, что и замененный, но требует наличия команды быстрого *умножения на 3*.

При подсчете количества единичных битов в слове по алгоритму, предложенному в HAKMEM [25, item 169], используется только три первых члена из формулы (1) для формирования трехбитовых слов, в каждом из которых помещается сумма его единичных битов. Затем значения, содержащиеся в соседних трехбитовых полях, складываются, а результат помещается в шестибитовое поле. После этого суммируются значения, содержащиеся в шестибитовых полях, вычисляя значение слова по модулю 63. На языке С этот алгоритм записывается следующим образом (обратите внимание, что длинные константы здесь записаны в восьмеричной системе счисления):

```
int pop(unsigned x)
{
    unsigned n;

    n = (x >> 1) & 033333333333;          // Подсчет битов
    x = x - n;                             // в 3-битовых
    n = (n >> 1) & 033333333333;          // полях
    x = x - n;                             //
    x = (x + (x >> 3)) & 030707070707;    // 6-разрядные суммы
    return x % 63;                         // Сложение 6-разрядных сумм
}
```

В последней строке используется беззнаковая функция получения значения x по модулю 63 (это может быть как знаковое, так и беззнаковое число, если длина слова кратна 3). Данная функция складывает значения, содержащиеся в 6-битовых полях слова x, что становится понятно, если рассматривать слово x как целое число, записанное в системе счисления с основанием 64. Остаток от деления целого числа в системе счисления с основанием b на $b-1$ для $b>3$ сравним по модулю $b-1$ с суммой цифр этого числа и, конечно, меньше b. Так как сумма цифр в нашем случае не превышает 32, значение $\text{modu}(x,63)$ должно быть равно сумме цифр x, т.е. количеству единичных битов в этом слове.

На компьютерах DEC PDP-10 есть команда, вычисляющая остаток от деления, второй операнд которой содержит адрес слова в памяти, а потому на этих машинах подсчет единичных битов в слове выполняется за 10 команд. На компьютере RISC требуется около 13 команд, если в их числе имеется отдельная команда *остатка от деления по модулю,* Однако, вероятно, это не очень быстрый метод, так как деление — операция довольно медленная. Простое расширение констант не позволяет работать с 64-битовыми словами, но вполне применимо для слов длиной до 62 бит.

В [26] предложена вариация алгоритма HAKMEM. Сначала с использованием формулы (1) параллельно подсчитывается количество единичных битов в восьми 4-битовых полях. Полученные 4-битовые значения преобразуются в 8-битовые суммы, а затем четыре байта суммируются при помощи умножения на число 0x01010101.

```
int pop(unsigned x)
{
   unsigned n;

   n = (x >> 1) & 0x77777777;          // Подсчет битов
   x = X - n;                           // в 4-битовых
   n = (n >> 1) & 0x77777777;          // полях
   x = X - n;
   n = (n >> 1) & 0X77777777;
   x = X - n;
   x = (x + (x >> 4)) & 0x0F0F0F0F;    // Вычисление сумм
   x = x*0x01010101;                    // Сложение байтов
   return x >> 24;
}
```

Здесь для вычислений требуется 19 базовых RISC-команд. Этот код эффективно работает на двухадресной машине, так как в первых шести строках выполняется только одна команда *пересылки регистра.* Неоднократно используемую маску 0x77777777 можно однократно загрузить в регистр и обращаться к ней при помощи команд, работающих только с регистрами. Кроме того, большинство сдвигов в этом коде — на один разряд.

Совсем другой метод подсчета битов предложен в [54, 60] (листинг 5.2). Здесь циклически сбрасывается крайний справа единичный бит исследуемого слова, до тех пор пока это слово не станет равным 0. Этот метод, для работы которого требуется выполнение $2+5\,\text{pop}(x)$ команд, эффективно работает со словами, в которых содержится только небольшое количество единичных битов.

Листинг 5.2. Подсчет единичных битов в малозаполненных словах

```
int pop(unsigned x)
{
   int n;

   n = 0;
```

```
    while (x != 0)
    {
        n = n + 1;
        X = X & (x - 1) ;
    }
    return n;
}
```

Для слов с большим количеством единичных битов применяется дуальный алгоритм. В слове циклически устанавливается крайний справа нулевой бит (x = x | (x+1)) до тех пор, пока во всех разрядах слова не оказываются единицы (т.е. пока слово не станет равным -1). После этого возвращается число $32-n$. (В других вариантах этого алгоритма биты исходного значения x могут быть инвертированы, начальное значение n может быть задано равным 32 и при вычислении уменьшаться, а не увеличиваться.)

Еще один интересный алгоритм подсчета единичных битов предложен в [47]. В этом алгоритме вычисляется сумма всех 32 слов, полученных в результате циклического сдвига слова влево на один разряд. Итоговая сумма равна значению $\text{pop}(x)$ со знаком минус! Следовательно:

$$\text{pop}(x) = -\sum_{i=0}^{31}\left(x \overset{rot}{\ll} i\right), \qquad (2)$$

где все сложения выполняются по модулю длины слова, а итоговая сумма рассматривается как целое, дополненное до 2. Этот способ — не более чем красивая, но не эффективная для практического применения формула: цикл выполняется 31 раз, т.е. требуется выполнение 63 команд, не считая команд управления циклом.

Чтобы понять, как работает уравнение (2), посмотрим, что происходит с каждым отдельным битом слова x. Этот бит при циклическом сдвиге оказывается в каждом разряде, и при сложении всех 32 чисел, полученных в результате 31 циклического сдвига, в итоговом слове в каждом разряде будет единица. Но это число представляет собой -1. Для иллюстрации рассмотрим 6-битовое слово $x = \mathbf{001001}$.

001001 x

010010 $x \overset{rot}{\ll} 1$

100100 $x \overset{rot}{\ll} 2$

001001 $x \overset{rot}{\ll} 3$

010010 $x \overset{rot}{\ll} 4$

100100 $x \overset{rot}{\ll} 5$

Разумеется, сдвиг вправо работает не менее корректно.

Метод подсчета единичных битов по формуле (1) очень похож на метод циклического сдвига и сложения, что становится понятно, если уравнение (1) переписать как

$$\text{pop}(x) = x - \sum_{i=1}^{31}\left(x \overset{u}{\gg} 1\right).$$

Вычисление функции $\text{pop}(x)$ по этой формуле будет происходить немного быстрее по сравнению с подсчетом единичных битов по формуле (2) по двум причинам. Во-первых, в нем используется команда сдвига вправо, которая, в отличие от команды циклического сдвига, есть практически на всех машинах; во-вторых, если при сдвиге слова получается нулевое значение, цикл завершается. Таким образом, это позволяет упростить цикл и сэкономить несколько итераций. Сравнение алгоритмов показано в листинге 5.3.

Листинг 5.3. Циклические алгоритмы подсчета количества единичных битов

```
int pop(unsigned x)
{
    int i, sum;

// Циклический сдвиг и сложение  // Сдвиг вправо и вычитание

    sum = x;                        // sum = x;
    for(i = 1; i <= 31; i++)        // while(x != 0)
    {                               // {
        x = rotatel(x,1);           //     x = x >> 1;
        sum = sum + x;              //     sum = sum - x;
    }                               // }
    return -sum;                    // return sum;
}
```

Алгоритм, вычисляющий функцию $\text{pop}(x)$ с использованием массива значений, менее оригинален по сравнению с предыдущими алгоритмами, тем не менее вполне может конкурировать с ними. Например, пусть для значений от 0 до 255 имеется массив `table` значений функции $\text{pop}(x)$. При подсчете единичных битов в слове выполняется четыре обращения к массиву `table` и затем складываются четыре полученных числа. Вот один из вариантов этого алгоритма (без условных переходов):

```
int pop(unsigned x)         // Поиск в таблице
{
    static char table[256] = {
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
        ...
        4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8};

    return table [x         & 0xFF] +
           table [(x >>  8) & 0xFF] +
           table [(x >> 16) & 0xFF] +
           table [(x >> 24)];
}
```

В [25, item 167] приведен краткий алгоритм подсчета количества единичных битов в 9-битовом значении, выровненном вправо и размещенном в регистре. Этот алгоритм работает только на машинах с длиной регистра 36 и больше бит. Ниже приведен вариант этого алгоритма для работы на 32-битовой машине (только для 8-битовых величин).

```
X = x * 0x08040201;     // Создаются 4 копии
x = x >> 3;             // Удаление соответствующих битов
x = x & 0x11111111;     // Каждый 4-й бит
x = x * 0x11111111;     // Сумма цифр (0 или 1)
x = x >> 28;            // Положение результата
```

Вот версия алгоритма для 7-битовых величин:

```
x = x * 0x02040810;     // Создаются 4 копии
x = x & 0x11111111;     // Каждый 4-й бит
x = x * 0x11111111;     // Сумма цифр (0 или 1)
x = x >> 28;            // Положение результата
```

Здесь последние два шага могут быть заменены вычислением остатка от x по модулю 15.

Эти методы работают недостаточно эффективно. Большинство программистов предпочитают при подсчете единичных битов пользоваться таблицами значений этой функции. Последний из рассмотренных алгоритмов, однако, имеет версию с использованием 64-битовой арифметики и может оказаться полезным на 64-битовых машинах, у которых есть команда быстрого умножения. Аргументом данного алгоритма является 15-битовая величина (я не думаю, что есть похожий алгоритм для 16-битовых величин, кроме случая, когда заранее известно, что не все 16 бит являются единичными). Тип данных long long представляет собой расширение языка С в компиляторе GNU [57] и означает целое число вдвое большей длины, чем тип int (в нашем случае тип long long имеет длину 64 бита). Суффикс ULL означает константы типа unsigned long long.

```
int pop (unsigned x)
{
    unsigned long long y;
    y = x * 0x0002000400080010ULL;
    y = y & OxI111111111111111ULL;
    y = y * OxI111111111111111ULL;
    y = y >> 60;
    return y;
}
```

Подсчет единичных битов в массиве

В простейшем способе подсчета единичных битов в массиве полных слов при отсутствии команды pop(x) сначала подсчитывается количество единичных битов в каждом слове массива, например с использованием процедуры из листинга 5.1, а затем полученные значения просто складываются.

Другой путь, который может оказаться более быстрым, состоит в использовании первых двух строк этой процедуры для групп из трех слов с последующим сложением трех промежуточных значений. Поскольку максимальное значение каждого промежуточного результата в каждом 4-разрядном поле равно 4, после сложения этих трех промежуточных результатов в каждом 4-разрядном поле будет максимум число 12, так что переполнения при сложении возникнуть не может. Затем каждая из полученных сумм может быть преобразована в слово с четырьмя 8-битовыми полями с максимальным значением поля, равным 24, с использованием

```
x = (X & OXOFOFOFOF) + ((x >> 4) & OxOFOFOFOF);
```

Полученные таким образом слова складываются до тех пор, пока максимальное значение их суммы в каждом поле результирующего слова остается меньше 255; всего можно сложить десять таких слов ($\lfloor 255/24 \rfloor$). После сложения 10 слов полученный результат преобразуется в слово с двумя 16-битовыми полями с максимальным значением 240 в каждом из них:

```
X = (X & OxOOFFOOFF) + ((x >> 8) & 0x00FF00FF);
```

Теперь, прежде чем потребуется применить очередное преобразование в 32-битовое слово, можно сложить 273 таких слова ($\lfloor 65535/240 \rfloor$). Это преобразование выполняется следующим образом:

```
X = (X & OXOOOOFFFF) + ((x >> 16);
```

На практике команды управления циклом существенно превысят полученную при использовании этого метода экономию, так что строго следовать описанной процедуре, вероятно, не имеет смысла. Например, код в листинге 5.4 использует только один промежуточный уровень. Сначала формируются слова с четырьмя 8-битовыми полями с промежуточными суммами в каждом поле. Затем все эти слова складываются с образованием целого слова, содержащего сумму. Количество слов с 8-битовыми полями, которое можно просуммировать без риска переполнения, равно $\lfloor 255/8 \rfloor = 31$.

Листинг 5.4. Подсчет единичных битов в массиве

```
int pop_array (unsigned A[], int n)
{

    int i, j, lim;
    unsigned s, s8, x;

    s = 0;
    for (i = 0; i < n; i = i + 31)
    {
        lim = min(n, i + 31);
        s8 = 0;
        for (j = i; j < lim; j++)
        {
            x = A[j];
            x = x - ((x >> 1) & 0x55555555);
            x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
            x = (x + (x >> 4)) & 0x0F0F0F0F;
            s8 = s8 + x;
        }
        x = (s8 & 0x00FF00FF) + ((s8 >> 8) & 0x00FF00FF);
        x = (x & 0x0000FFFF) + (x >> 16);
        s = s + x;
    }
    return s;
}
```

Интересно сравнить этот алгоритм с простым циклическим алгоритмом. Обе процедуры были откомпилированы GCC на одной машине, очень похожей на RISC с основным набором команд. Результат — 22 команды на одно слово для простого циклического метода и 17.6 команды на слово по алгоритму из листинга 5.4, т.е. экономия на слово составляет 20%.

Применение

Функция $\text{pop}(x)$ используется, например, при вычислении "расстояния Хемминга" (Hamming distance) между двумя битовыми векторами. Расстояние Хемминга (концепция из теории кодов с исправлением ошибок) представляет собой количество разрядов, в которых эти векторы отличаются, т.е.

$$\text{dist}(x, y) = \text{pop}(x \oplus y).$$

(Смотрите, например, главу о кодах с исправлением ошибок в [12].)

Другое применение — обеспечение быстрого индексного доступа к элементам разреженного массива *A*, представленного некоторым компактным образом. В компактном представлении сохраняются только определенные, ненулевые элементы массива. Пусть

есть вспомогательный массив битовых строк *bits* (содержащих 32-разрядные слова), имеющий единичный бит для каждого индекса i, для которого определен элемент $A[i]$. Для ускорения доступа имеется также массив слов ***bitsum***, такой, что $bitsum[j]$ представляет собой общее количество во всех словах массива *bits* единичных битов, предшествующих j-му. Проиллюстрируем сказанное на массиве, в котором определены 0, 2, 32, 47, 48 и 95-й элементы.

bits	*bitsum*	*Данные*
0x00000005	О	$A[0]$
0x00018001	2	$A[2]$
0x80000000	5	Л [32]
		A[47]
		A[48]
		A(95]

Для заданного индекса i, $0 < i < 95$, соответствующий индекс *sparse_i* в массиве данных задается через количество единичных битов элементов массива *bits,* предшествующих биту, соответствующему *i.* Его можно вычислить следующим образом:

```
j = i >> 5;                       // j = i/32
k = i & 31;                       // k = rem(i,32)
mask = 1 << k;                    // "1" в разряде k
if ((bits[j] & mask) == 0)
        goto no_such_element;
mask = mask - 1;                  // единицы справа от k
sparse_i =
        bitsum[j] + pop(bits[j] & mask);
```

Затраты памяти при таком представлении разреженного массива — два бита на элемент полного массива.

Функция $\text{pop}(x)$ применяется также при вычислении количества завершающих нулей слова (см. раздел 5.4).

5.2. Четность

Под "четностью" (parity) битовой строки понимается, какое количество единичных битов — четное или нечетное — содержит эта строка. Строка считается "нечетной", если она содержит нечетное количество единичных битов; в противном случае строка является четной[1].

Вычисление четности слова

Результат проверки на четность равен 1, если некоторое слово *x* является нечетным, и равен 0, если слово является четным. Этот результат представляет собой сумму всех битов слова по модулю 2, т.е. над всеми битами *x* выполняется операция *исключающего или.*

---

[1] В данном разделе значения терминов "четное" и "нечетное" отличаются от обычных. Везде в разделе, где не оговорено иное, "четность" означает наличие четного числа единичных битов в рассматриваемом значении, а "нечетность" — нечетное количество битов. — *Прим. ред.*

При проверке слова на четность можно использовать функцию $\text{pop}(x)$ — в таком случае четность представляет собой младший бит значения, возвращаемого этой функцией. Этот метод вполне применим при наличии команды для вычисления $\text{pop}(x)$. Если же такой команды нет, то вместо использования кода для вычисления $\text{pop}(x)$ лучше воспользоваться более эффективными специализированными методами.

Один из таких методов состоит в вычислении

$$y \leftarrow \bigoplus_{i=0}^{n-1} \left(x \overset{u}{\gg} i \right),$$

где n — длина слова, а результат проверки на четность слова x помещается в младший бит y. (Оператор $\oplus$ означает *исключающее или,* но в данном случае его можно заменить обычным сложением.)

Ниже приводится более быстрый способ проверки на четность для не слишком больших значений и (в примере л $= 32$; сдвиги могут быть как знаковыми, так и беззнаковыми).

```
У = Х ^ (Х >> 1);
у = у ^ (у >> 2);
у = у ^ (у >> 4);                 (3)
у = у ^ (у >> 8);
У = У ^ (у >> 16);
```

Всего выполняется 10 команд, что значительно меньше, чем в первом методе (62 команды), даже при полном развертывании цикла. Здесь бит четности также оказывается в младшем разряде y. В действительности всегда при выполнении беззнаковых сдвигов i-й бит результата y дает четность битовой строки, состоящей из всех битов x от i-го до старшего. Кроме того, так как *исключающее или* представляет собой свою собственную инверсию, четность $(x \gg i)$ Ф $(x \gg j)$ равна четности подслова, состоящего из битов x су по $\iota - 1$ $(\iota \geq$ у).

Приведенный выше метод является примером метода "параллельного префикса", или "сканирования", который применяется в параллельных вычислениях [30, 41]. При наличии достаточного количества процессоров он позволяет преобразовать некоторые кажущиеся последовательными процессы в параллельные, сокращая время вычислений от $O(n)$ до $O(\log n)$. Например, если необходимо выполнить операцию поразрядного *исключающего или* над целым массивом, вы можете сначала выполнить действия (3) над каждым словом массива и затем применить, по сути, ту же методику к массиву, выполняя операцию *исключающего или* над словами массива. При таком подходе выполняется большее количество элементарных (с отдельными словами) команд *исключающего или,* чем в простом алгоритме, работающем слева направо, а следовательно, для однопроцессорного компьютера применять данный метод не стоит. Но на компьютере с достаточным количеством параллельных процессоров проверка на четность по описанному выше алгоритму занимает время $O(\log n)$, а не $O(n)$ (где n — количество слов в массиве).

Непосредственным приложением метода (3) является преобразование целых чисел в код Грея (см. главу 13).

Если изменить код (3), заменив все сдвиги вправо сдвигами влево, то результат проверки на четность целого слова x отображается в старшем бите y, а i-й бит слова y содержит четность битов слова x, расположенных в i-й позиции и *правее* нее.

При использовании *циклических сдвигов* в результирующем слове во всех разрядах будут либо единицы, если x содержит нечетное количество единичных битов, либо нули, если их количество четное.

В следующем методе четность слова x вычисляется за девять команд, при этом результат оказывается равен 0 или 1 (все сдвиги в коде беззнаковые).

```
x = x ^ (x >>1) ;
x = (x ^ (x >> 2)) & 0x11111111;
x = x*0x11111111;
p = (x >>28) & 1;
```

После выполнения второй строки каждая шестнадцатеричная цифра в x примет значение 0 или 1, в зависимости от четности исходной цифры. Команда *умножения* суммирует все цифры, помещая сумму в старший шестнадцатеричный разряд. Переносов при этом возникнуть не может, поскольку максимальное значение любого байта равно 8.

Команды *умножения* и *сдвига* можно заменить командой, которая вычисляет остаток от деления x на 15, что дает (медленную) реализацию алгоритма за восемь команд, если, конечно, в компьютере есть команда получения *остатка от деления на непосредственно заданное значение*.

Добавление бита четности к 7-битовой величине

В [25, item 167] есть интересный алгоритм, добавляющий бит четности к 7-битовой величине (выровненной вправо и помещенной в регистр). Под этим подразумевается установка бита четности слева от семи битов таким образом, чтобы получить восьмибитовую величину с четным количеством единичных битов. В указанной работе приведен пример кода для 36-битовой машины, но он корректно работает и на 32-битовых машинах.

$$\operatorname{modu}\left(\left(x * \mathbf{0x10204081}\right) \,\&\, \mathbf{0x888888FF}, 1920\right).$$

Здесь $\operatorname{modu}(a,b)$ означает остаток от беззнакового деления a на b, аргументы и результат интерпретируются как целые числа без знака, "$*$" означает умножение по модулю 2^{32}, а константа 1920 равна $15 \cdot 2^7$. Фактически здесь вычисляется сумма битов в слове x с размещением этой суммы слева от семи битов x. Например, это выражение преобразует число **0x0000007F** в **0x000003FF** и **0x00000055** в **0x00000255**.

В [25] есть еще одна остроумная формула для добавления битов четности к 7-битовому числу (при этом образуется 8-битовое число с нечетным количеством единичных битов):

$$\operatorname{modu}\left(\left(x * \mathbf{0x00204081}\right) \mid \mathbf{0x3DB6DB00}, 1152\right),$$

где $1152 = 9 \cdot 2^7$. Понять, как работает эта формула, поможет тот факт, что степень 8 по модулю 9 равна ± 1. Если заменить **0x3DB6DB00** значением **0xBDB6DB00**, эта формула будет давать восьмибитовое число с четным количеством единичных битов.

На современных машинах все эти методы недостаточно практичны, так как при том, что память дешевеет, деление все еще выполняется довольно медленно. Поэтому большинство программистов предпочитают использовать вместо вычислений поиск в таблице.

Применение

Операция четности используется при перемножении битовых матриц в GF(2) (где операция *сложения* представляет собой *исключающее или*).

5.3. Подсчет ведущих нулевых битов

Существует ряд простых способов подсчета ведущих нулевых битов слова, реализуемых при помощи метода бинарного поиска. Ниже приведена модель, имеющая ряд вариаций. На процессоре с базовым набором RISC-команд вычисления выполняются за 20–29 команд. Все сравнения беззнаковые.

```
if (x == 0) return(32);
n = 0;
if (x <= 0x0000FFFF) {n = n +16; x = x <<16;}
if (x <= 0x00FFFFFF) {n = n + 8; x = x << 8;}
if (x <= 0x0FFFFFFF) {n = n + 4; x = x << 4;}
if (x <= 0x3FFFFFFF) {n = n + 2; x = x << 2;}
if (x <= 0x7FFFFFFF) {n = n + 1;}
return n;
```

В другом варианте все сравнения заменяются командами *и:*

```
if ((x & 0xFFFF0000) == 0) {n = n +16; x = x <<16;}
if ((x & 0xFF000000) == 0) {n = n + 8; x = x << 8;}
...
```

Еще один вариант, позволяющий избежать больших непосредственно задаваемых значений, использует команды *сдвига вправо.*

Последний из операторов `if` просто прибавляет единицу к n, если старший бит x равен 0, так что в варианте без использования условного перехода можно прибегнуть к инструкции

```
n = n + 1 - (x >> 31);
```

Здесь "+1" можно опустить, если инициализировать переменную n не 0, а 1. Это наблюдение приводит к алгоритму, для работы которого потребуется выполнение от 12 до 20 основных RISC-команд (листинг 5.5). Этот код можно улучшить еще немного, если x начинается с единичного бита; тогда первую строку можно заменить следующей:

```
if ((int)x <= 0) return (~x >> 26) & 32;
```

Листинг 5.5. Подсчет ведущих нулевых битов бинарным поиском

```
int nlz(unsigned x)
{
    int n;

    if (x == 0) return(32);
    n = 1;
    if ((x >> 16) == 0) {n = n +16; x = x <<16;}
    if ((x >> 24) == 0) {n = n + 8; x = x << 8;}
    if ((x >> 28) == 0) {n = n + 4; x = x << 4;}
    if ((x >> 30) == 0) {n = n + 2; x = x << 2;}
    n = n - (x >> 31);
    return n;
}
```

В листинге 5.6 показан вариант подсчета одним из методов, обратных приведенному выше. Он требует выполнения тем меньшего количества операций, чем больше ведущих нулей имеется в слове, и позволяет избежать использования больших непосредственно задаваемых чисел и больших сдвигов. Всего выполняется от 12 до 20 основных RISC-команд.

Листинг 5.6. Подсчет ведущих нулевых битов бинарным поиском в обратном направлении

```
int nlz(unsigned x)
{
    unsigned y;
    int n;

    n  = 32;
    y = X >>16;  if (y != 0) {п = п -16;  x m y;}
    y = x >> 8;  if (y != 0) {п = п - 8; x = y;}
    y = x >>4;   if (y != 0) {п = п - 4; x = y;}
    y = x >>2;   if (y !- 0) {п = п - 2; x = y;}
    y = X >> 1;  if (y != 0) return n - 2;
    return n - x;
}
```

Этот алгоритм позволяет использовать метод поиска в таблице, при этом последние четыре исполняемые строки кода заменяются строками

```
static char table[256] = {0,1,2,2,3,3,3,3,4,4,..., 8);
return n - table[x];
```

Использовать вспомогательный поиск в таблице позволяют многие алгоритмы, хотя упоминается об этом нечасто.

Для компактности два последних алгоритма можно кодировать с использованием циклов. Например, алгоритм из листинга 5.7 является вариантом алгоритма из листинга 5.6 с использованием цикла. Выполняется этот вариант алгоритма за 23–33 базовые RISC-команды, 10 из которых представляют собой команды условного ветвления.

Листинг 5.7. Подсчет ведущих нулевых битов бинарным поиском с использованием цикла

```
int nlz(unsigned x)
{
    unsigned y;
    int n, c;

    n =32;
    C = 16;
    do {
        y = X >> C;
        if (y != 0) {n = n - c; x = y;}
        C = C >> 1;
    } while (c != 0);
   n = n - (x >> 31) ;
   return n - x;
}
```

Очевидно, для подсчета количества ведущих нулей можно циклически выполнять сдвиг влево на 1, подсчитывая количество нулевых битов до тех пор, пока знаковый бит не станет равным 1; можно также циклически сдвигать слово на одну позицию вправо до тех пор, пока все биты слова не окажутся нулевыми. Оба алгоритма достаточно компактны и хорошо работают со словами, содержащими малое (или соответственно большое) количество ведущих нулевых битов. Можно объединить описанные методы в одном алгоритме, как показано в листинге 5.8. Способ объединения двух алгоритмов и получения

результата по завершении вычислений одним из них упомянут здесь потому, что имеет множество применений. В частности, он позволяет получить код, быстро работающий на суперскалярных и VLIW-машинах (машинах со сверхбольшой длиной слова) вследствие наличия соседних независимых команд (эти машины позволяют выполнять несколько команд одновременно при условии их независимости).

Листинг 5.8. Двухсторонний подсчет ведущих нулевых битов

```
int nlz(int x)
{
    int y, n;

    n = 0;
    y = X;
L:  if (x < 0) return n;
    if (y == 0) return 32 - n;
    n = n + 1;
    X = X << 1;
    y = y >> 1;
    goto L;
}
```

На машине с базовым набором RISC-команд этот код выполняется за $\min(3 + 6\,\text{nlz}(x), 5 + 6(32 - \text{nlz}(x)))$ команд, что в худшем случае равно 99 командам. Однако можно представить суперскалярную или VLIW-машину, в которой тело цикла выполняется за один такт (если результат сравнения получается как побочный результат выполнения команд сдвига; в противном случае потребуется два такта), плюс накладные расходы на команды ветвления.

Достаточно просто преобразовать код из листингов 5.5 и 5.6 в эквивалентный, но без использования условных переходов. В листинге 5.9 представлен метод вычисления функции nlz за 28 основных RISC-команд.

Листинг 5.9. Подсчет ведущих нулевых битов бинарным поиском без условных переходов

```
int nlz(unsigned x)
{
    int y, m, n;

    y = -(x >> 16);      // Если левая половина x = 0,
    m = (y >> 16) & 16;  // п = 16. Если левая половина
    п = 16 - т;          // не нулевая, установить п = 0 и
    x = x >> т;          // сдвинуть x вправо на 16 бит.
                         // x принимает вид 0000xxxx
    y = x - 0x100;       // Если разряды 8-15 нулевые,
    m = (y >> 16) & 8;   // к п добавляется 8 и выполняется
    n = n + m;           // сдвиг x влево на 8
    X = X << т;

    y = x - 0x1000;      // Если разряды 12-15 нулевые,
```

```
    m = (y >> 16) & 4;    // увеличиваем п на 4 и
    n = n + m;            // сдвигаем x влево на 4
    x = x << т;

    y = x - 0x4000;        // Если разряды 14-15 нулевые,
    т = (y >> 16) & 2;     // увеличиваем n на 2 и
    n = n + m;             // сдвигаем x влево на 2
    x = x << т;

    y = x >> 14;           // Устанавливаем y = 0, 1, 2 или 3
    m = y & ~(y >> 1);     // m = O, 1, 2 или 2 соответственно
    return n + 2 - m;
}
```

Если ваш компьютер оснащен командой для вычисления функции $\text{pop}(x)$, то существует эффективный способ подсчета ведущих нулевых битов, показанный в листинге 5.10. Пять присвоений значений переменной x в действительности можно выполнять в любом порядке. Этот алгоритм выполняется за 11 команд и не использует команд ветвления. Он полезен даже в том случае, когда специальной команды для вычисления функции $\text{pop}(x)$ в компьютере нет. В этой ситуации можно использовать программу из листинга 5.1, которая выполняется за 21 команду, так что поиск количества ведущих нулевых битов выполняется при помощи 32 базовых RISC-команд, причем среди них нет ни одной команды условного перехода.

Листинг 5.10. Подсчет ведущих нулевых битов с использованием функции $\text{pop}(x)$

```
int nlz(unsigned x)
{
    int pop(unsigned x);

    x = x | (x >>1);
    x = x | (x >> 2);
    x = x | (x >>4);
    x = x | (x >>8);
    x = x | (x >>16);
    return pop(~x);
}
```

Методы с плавающей точкой

При подсчете ведущих нулевых битов слова можно использовать нормализованные числа с плавающей точкой, например числа с плавающей точкой в IEEE-формате. Идея заключается в преобразовании данного беззнакового целого числа в число с плавающей точкой двойной точности, выделении из него степенной части и вычитании ее из константы. Соответствующий код приведен в листинге 5.11.

Листинг 5.11. Подсчет ведущих нулевых битов с использованием чисел с плавающей точкой в IEEE-формате

```
int nlz(unsigned k)  {
    union {
        unsigned asInt[2];
        double asDouble;
    };
    int n;
```

```
    asDouble = (double)k + 0.5;
    n = 1054 - (asInt[LE] >> 20);
    return n;
}
```

Этот код использует анонимное (безымянное) объединение С++ для перекрытия целого числа и величины с плавающей точкой двойной точности. Переменная LE должна быть равна 1, если процедура выполняется на машинах, где адрес младшего байта меньше адреса старшего, и 0 в противном случае. Слагаемое 0.5 (или другое малое число) необходимо для корректной работы программы при k = 0.

Оценить время выполнения этой программы практически невозможно, так как на разных машинах числа с плавающей точкой обрабатываются по-разному. Например, на многих машинах, кроме регистров для хранения целых чисел, имеются специальные регистры для хранения чисел с плавающей точкой. На таких машинах, в частности, может потребоваться пересылка данных в память для преобразования целого числа в число с плавающей точкой и наоборот.

Код в листинге 5.11 не соответствует ANSI-стандарту языков С и С++, поскольку он обращается к одной и той же области памяти как к двум различным типам данных. Следовательно, нет никаких гарантий, что этот код будет правильно работать на всех машинах или со всеми компиляторами. Тем не менее он работает с компилятором XLC IBM на платформе AIX и с компилятором GCC на платформах AIX и Windows 2000 при всех уровнях оптимизации (по крайней мере на момент написания книги). Если изменить код как показано ниже, то код в режиме оптимизации в указанных системах работать не станет.

```
xx = (double)k + 0.5;
n = 1054 - (*((unsigned *)&xx + LE) >> 20);
```

Кстати, этот код нарушает еще один стандарт ANSI, а именно то, что арифметические действия могут выполняться только с указателями на элементы массива [8]. Проблема, тем не менее, связана с первым нарушением стандарта.

Несмотря на то что данный код не совсем корректен[2], далее приведено несколько его вариантов.

```
asDouble = (double)k;
n = 1054 - (asInt[LE] >> 20);
n = (n & 31) + (n >> 9);


k = k & ~(k >> 1);
asFloat = (float)k + 0.5f;
n = 158 - (asInt >> 23);


k = k & ~(k >> 1);
asFloat = (float)k;
n = 158 - (asInt >> 23);
n = (n & 31) + (n >> 6);
```

[2] Некорректен в силу способа использования языка программирования С. Эти методы совершенно корректно будут работать при кодировании на машинном языке либо будучи сгенерированными компилятором для конкретного типа компьютера.

В первом варианте проблема с k = о решена не путем добавления слагаемого 0.5, а при помощи дополнительных арифметических действий над результатом п (которое без применения коррекции будет равно 1054 (0x41E)).

В следующих двух вариантах этого кода используются числа с плавающей точкой одинарной точности, которые преобразуются обычным методом с использованием анонимного объединения. Правда, здесь возникает новая проблема: при округлении результата возможны неточности, как при округлении к ближайшему числу (наиболее распространенный метод округления), так и при округлении в большую сторону. В режиме округления к ближайшему числу проблемы с округлением k возникают в диапазонах (в шестнадцатеричной записи) от FFFFFF80 до FFFFFFFF, от 7FFFFFCO до 7FFFFFFF, от 3FFFFFEO до 3FFFFFFF и т.д. При округлении этих чисел добавление единицы приводит к появлению переноса в левых разрядах, что влечет за собой изменение позиции старшего единичного бита. Показанные выше способы коррекции сбрасывают бит справа от старшего единичного бита, позволяя тем самым избежать переноса.

Компилятор GNU C/C++ дает возможность кодировать любой из этих методов в виде макроса, позволяя вместо вызова функции использовать встроенный код [57]. Компилятор разрешает использовать в макросе объявления, а последнее выражение среди операторов макроса дает возвращаемое макросом значение. Ниже приводится пример макроопределения для алгоритма с применением чисел с одинарной точностью. (В языке С в названиях макросов обычно используются прописные буквы.)

```
#define NLZ(kp) \
    ({union {unsigned _asInt; float _asFloat ; } ;    \
       unsigned _k = (kp), _kk = _k & ~ (_k >> 1) ;  \
       _asFloat = (float)_kk + 0.5f;                 \
       158 - (_asInt >> 23) ; })
```

Символы подчеркивания в именах нужны для того, чтобы избежать конфликтов имен с параметром kp (обычно определенные пользователем имена переменных не начинаются с символа подчеркивания).

Связь с логарифмом

По существу, nlz представляет собой функцию "целого логарифма по основанию 2". Для беззнакового $x \neq 0$ справедливы следующие соотношения (см. также раздел 11.4):

$$\lfloor \log_2(x) \rfloor = 31 - \mathrm{nlz}(x),$$
$$\lceil \log_2(x) \rceil = 32 - \mathrm{nlz}(x-1).$$

Другой тесно связанной с ней функцией является функция *bitsize* — количество битов, которое требуется для представления ее аргумента как знаковой величины в дополнительном коде. Эта функция определяется следующим образом:

$$\mathrm{bitsize}(x) = \begin{cases} 1, & x = -1 \text{ или } 0, \\ 2, & x = -2 \text{ или } 1 \\ 3, & -4 \le x \le -3 \text{ или } 2 \le x \le 3, \\ 4, & -8 < x < -5 \text{ или } 4 < x < 7, \\ \dots & \dots \\ 32, & -2^{31} \le x \le -2^{30}+1 \text{ или } 2^{30} \le x \le 2^{31}-1. \end{cases}$$

Из определения очевидно, что $\text{bitsize}(x) = \text{bitsize}(-x-1)$. Но, поскольку $-x-1 = \neg x$, можно записать следующий код для вычисления функции bitsize (сдвиг в коде — знаковый):

```
X = X ^ (x >> 31);        // Если (x < 0), X = -x - 1;
return 33 - nlz(x);
```

Применение

Функции вычисления ведущих нулевых битов слова обычно используются при моделировании арифметических действий над числами с плавающей точкой и в различных алгоритмах деления (см. листинги 9.1 и 9.3). Кроме этого, команда $\text{nlz}(x)$ имеет множество других применений.

Она может использоваться при вычислении отношения $x = y$ за три команды (см. раздел 2.11) и ряда элементарных функций (см. разделы 11.1–11.4).

Еще одно интересное применение функции — для генерации экспоненциально распределенных случайных целых чисел. Для этого генерируются случайные равномерно распределенные целые числа и к результату применяется функция nlz [19]. Вероятность возврата функцией значения 0 равна 1/2, вероятность значения 1 равна 1/4, 2 — 1/8 и т.д. Кроме того, функция применяется в качестве вспомогательной при поиске в слове подстроки определенной длины, состоящей из единичных (или нулевых) битов (процесс, используемый в некоторых алгоритмах распределения дискового пространства). В последних двух задачах используется также функция, вычисляющая *количество завершающих нулевых битов*.

5.4. Подсчет завершающих нулевых битов

Если имеется команда подсчета ведущих нулевых битов, то подсчет количества завершающих нулевых битов лучше всего свести к использованию этой команды:

$$32 - \text{nlz}(\neg x \,\&\, (x-1)).$$

При наличии команды вычисления количества единичных битов слова существует более эффективный метод, состоящий в формировании маски для выделения всех завершающих нулевых битов и подсчете единиц в этой маске [27], например так:

$$\text{pop}(\neg x \,\&\, (x-1)) \text{ или}$$
$$32 - \text{pop}(x \mid -x).$$

Сформировать маску для выделения завершающих нулевых битов можно по формулам, которые приводятся в разделе 2.1. Эти методы применимы даже в случае, когда среди команд компьютера команда подсчета количества единичных битов слова отсутствует. Например, если для подсчета единичных битов воспользоваться алгоритмом из листинга 5.1, то первое из приведенных выше выражений вычисляется за $3 + 21 = 24$ команды, среди которых нет команд ветвления.

В листинге 5.12 показан алгоритм, который выполняет описанные действия непосредственно, требуя от 12 до 20 базовых RISC-команд (для $x \neq 0$).

Листинг 5.12. Подсчет завершающих нулевых битов бинарным поиском

```
int ntz(unsigned x)
{
    int n;

    if (x == 0) return(32);
    n = 1;
    if ((X & 0x0000FFFF) == 0) {n = n +16; x = X >>16;}
    if ((x & 0x000000FF) == 0) {n = n + 8; X = x >> 8;}
    if ((X & 0x0000000F) == 0) {n = n + 4; X = x >> 4;}
    if ((x & 0x00000003) == 0) {n = n + 2; X = X >> 2;}
    return n - (x & 1);
}
```

Выражение n + 16 можно упростить до 17, если компилятор недостаточно интеллектуален, чтобы сделать это самостоятельно (это не влияет на подсчитанное нами количество команд).

Еще один вариант представлен в листинге 5.13. Здесь используются малые непосредственно задаваемые значения и выполняются более простые действия. При вычислениях требуется выполнить от 12 до 21 базовой RISC-команды. В отличие от первой процедуры, при меньшем количестве завершающих нулей выполняется большее количество команд (но, впрочем, и большее количество условных переходов, пропускающих вычисления).

Листинг 5.13. Подсчет завершающих нулевых битов, малые непосредственные значения

```
int ntz(unsigned x)
{
    unsigned y;
    int n;

    if (x == 0) return 32;
    n = 31;
    y = X <<16; if (y != 0) {n = n -16; X = y;}
    y = x << 8; if (y != 0) {n = n - 8; X = y; }
    y = x << 4; if (y != 0) {n = n - 4; x = y;}
    y = x << 2; if (y != 0) {n = n - 2; x = y;}
    y = x << 1; if (y != 0) {n = n - 1;}
    return n;
}
```

Строку перед оператором return можно заменить строкой, которая экономит один условный переход (но не команду):

```
n = n - ((x << 1) >> 31);
```

В смысле количества выполняемых команд трудно превзойти поиск по дереву [4]. В листинге 5.14 представлена соответствующая процедура для 8-битового аргумента. В каждом пути этой процедуры выполняется семь команд (за исключением последних двух — return 7 и return 8, где выполняется девять команд). Для 32-битового аргумента требуется выполнить от 11 до 13 команд. К сожалению, для больших размеров слов программа быстро становится все больше и больше. Так, для 8-битового аргумента исходный код занимает 12 исполняемых строк (и компилируется в 41 команду); 32-битовый аргумент требует 48 исполняемых строк кода и 164 команды, а 64-битовый — еще вдвое больше.

Листинг 5.14. Подсчет завершающих нулевых битов, поиск по бинарному дереву

```
int ntz(char x)
{
    if (x & 15)  {
        if (x & 13)  {
            if (x & 1) return 0;
            else return 1;
        }
        else if (x & 4) return 2;
        else return 3;
    }
    else if (x & 0x30)  {
        if (x & 0x10) return 4;
        else return 5;
    }
    else if (x & 0x40) return 6;
    else if (x) return 7;
    else return 8;
}
```

Если ожидается малое или, напротив, большое число завершающих нулевых битов, то лучше использовать простые циклические алгоритмы (листинг 5.15). Код в левой части листинга выполняется за $5 + 3\,\mathrm{ntz}(x)$ команд, а в правой — за $3 + 3(32 - \mathrm{ntz}(x))$ базовых RISC-команд.

Листинг 5.15. Подсчет завершающих нулевых битов с использованием циклов

```
int ntz(unsigned x)
{
    int n;

    X = -X & (X - 1) ;
    n = 0;                     // n = 32;
    while(x != 0) {            // while (x != 0)  {
        n = n + 1;             //     n» n - 1;
        X = X >> 1;            //     X = X + X;
   }                           // }
   return n;                   // return n;
}
```

Интересно отметить, что при равномерном распределении чисел * среднее количество завершающих нулевых битов очень близко к 1.0. Чтобы увидеть это, сложим произведения $p_i n_i$, где p_i — вероятность того, что в слове содержится n_i завершающих нулевых битов. Таким образом

$$S \cong \frac{1}{2}\cdot 0+\frac{1}{4}\cdot 1+\frac{1}{8}\cdot 2+\frac{1}{16}\cdot 3+\frac{1}{32}\cdot 4+\frac{1}{64}\cdot 5+\ldots \cong \sum_{n=0}^{\infty}\frac{n}{2^{n+1}}.$$

Чтобы вычислить эту сумму, рассмотрим следующий массив:

$$\begin{array}{cccccc} 1/4 & 1/8 & 1/16 & 1/32 & 1/64 & \ldots \\ & 1/8 & 1/16 & 1/32 & 1/64 & \ldots \\ & & 1/16 & 1/32 & 1/64 & \ldots \\ & & & 1/32 & 1/64 & \ldots \\ & & & & 1/64 & \ldots \\ & & & & & \ldots \end{array}$$

Сумма каждого столбца является членом ряда S. Следовательно, S — это сумма всех элементов массива. Вычислим сумму каждой строки.

$$\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\ldots=\frac{1}{2}$$
$$\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\frac{1}{64}+\ldots=\frac{1}{4}$$
$$\frac{1}{16}+\frac{1}{32}+\frac{1}{64}+\frac{1}{128}+\ldots=\frac{1}{8}$$
$$\ldots$$

Полная сумма, таким образом, равна $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots=1$. Абсолютная сходимость исходного ряда оправдывает использование перестановки.

Иногда требуется функция, аналогичная $\mathrm{ntz}(x)$, но рассматривающая аргумент, равный 0, как специальный (возможно, ошибочный), так что значение функции в этой точке должно отличаться от "обычных" значений. Например, определим функцию "количество множителей 2 в x" как

$$\mathrm{nfact2}(x)=\begin{cases}\mathrm{ntz}(x), & x \neq 0, \\ -1, & x=0.\end{cases}$$

Эту функцию можно вычислить по формуле $31 - \mathrm{nlz}(x \;\&\; -x)$.

Применение

В [19] рассматривается несколько интересных применений функции, вычисляющей количество завершающих нулевых битов. Ее называют "функцией линейки" *(rulerfunction),* так как она задает отметки на линейке, которая разделена на половины, четверти, восьмые части и т.д.

Функция ntz используется в алгоритме Госпера (R.W. Gosper), обнаруживающем циклы и заслуживающем детального рассмотрения в силу своей элегантности и функциональности, которую на первый взгляд от него трудно ожидать.

Пусть имеется последовательность X_0, X_1, $X_2,\ldots$, определяемая правилом $X_{n+1}=f(X_n)$. Если функция f ограничена, то эта последовательность является периодической. Она состоит из некоторой ведущей последовательности X_0, X_1, X_2, ..., $X_{\mu-1}$, за которой идет периодически повторяющаяся последовательность X_μ, $X_{\mu+1}$, ..., $X_{\mu+\lambda-1}$ ($X_\mu=X_{\mu+\lambda}X_{\mu+1}=X_{\mu+\lambda-1}$ и т.д., где λ — ее период). Нам требуется найти индекс первого элемента периодической подпоследовательности μ и период Я. Обнаружение цикла часто используется при тестировании генераторов случайных чисел и для определения, имеется ли цикл в связанном списке.

Конечно, можно сохранять все значения последовательности по мере их вычисления и сравнивать новый элемент со всеми предыдущими. Таким образом можно непосредст-

венно вычислить начало второго цикла. Однако имеются и более эффективные алгоритмы — как в смысле времени выполнения, так и в смысле необходимой памяти.

Возможно, самый простой метод предложен **Флойдом** (R.W. Floyd) [39, раздел. 3.1, задача 6]. В этом алгоритме итеративно выполняется следующий процесс:

$$x = f(x)$$
$$y = f(f(y))$$

(где начальные значения x и y равны X_0). После л-го шага $x=X_n$ и $y=X_{2n}$. Затем эти значения сравниваются. Если они равны, значит, элементы последовательности X_n и X_{2n} отделены друг от друга количеством элементов, кратным периоду Я, т.е. $2n-n=n$ кратно Я. Чтобы найти индекс μ, последовательность генерируется заново и X_0 сравнивается с X_n, X_1 — с X_{n+1} и так до тех пор, пока не будет найден элемент X_μ, равный $X_{n+\mu}$. И наконец, Я определяется путем дальнейшей генерации элементов и сравнением X_μ с элементами $X_{\mu+1}, X_{\mu+2}, \ldots$. Этот алгоритм требует небольшого количества памяти, но **функция f при** этом вычисляется многократно.

В алгоритме Госпера [25, item 132; 39, Ответы на упражнения к разделу 3.1, задача 7] определяется только период Я, но не начальная точка μ первого цикла. Главное достоинство данного алгоритма в том, что в нем не требуется повторное вычисление функции/, а также скорость работы и экономное использование памяти. Алгоритм не ограничен какими-либо значениями; для работы ему требуется таблица размера $\log_2(Л)+1$, где Л — максимально возможный период. Это не так много: например, если заранее известно, что $Л < 2^{32}$, то достаточно массива из 33 слов.

В листинге 5.16 приведен код на языке С, реализующий алгоритм Госпера. В качестве параметров функции передается указатель на рассматривавшуюся функцию / и начальное значение X_0. Функция возвращает верхнюю и нижнюю границы μ и период Я. (Хотя алгоритм Госпера и не позволяет определить точное значение μ, он позволяет вычислить его нижнюю и верхнюю границы μ_l и μ_u, такие, что $\mu_u - \mu_l + 1 < \max(Я-1,1)$.) Алгоритм работает путем сравнения X_n (для n = 1, 2, ...) с подмножеством (размером $\lfloor \log_2 n \rfloor + 1$) предшествующих значений. Элементами данного подмножества являются ближайшие предшествующие X/, такие, что $\iota+1$ заканчивается единичным битом (т.е. i — четное число, предшествующее и), $\iota+1$ заканчивается ровно одним нулевым битом, $\iota+1$ заканчивается ровно двумя нулевыми битами и т.д.

Листинг 5.16. Алгоритм Госпера

```
void ld_Gosper(int (*f) (int), int XO, int *mu_l,
               int *mu_u, int *lambda)
{
    int Xn, k, m, kmax, n, lgl;
    int T[33];

    T[0] = XO;
    Xn = XO;
    for (n = 1; ; n++)
    {
        Xn = f(Xn);
        kmax = 31 - nlz(n);          // Floor(log2 n)
        for (k = 0; k <= kmax; k++)
        {
            if (Xn == T[k]) goto L;
        }
```

```
        T[ntz(n+1)] = Xn;              // Нет совпадений
    }
L:
    //Вычисляем τ = max{i | i < n и ntz(i+1) = k}

    m = ((((n>> k) - 1) | 1) << k) - 1;
    *lambda = n - m;
    lgl = 31 - nlz(*lambda - 1);       // Ceil(log2 lambda)-1
    *mu_u = m;                        // Верхняя граница mu
    *mu_l = m - max(1, 1 << lgl) + 1; // Нижняя граница mu
}
```

Таким образом, сравнения выполняются так:

$X_1 : X_0$	$X_7 : X_6, X_5, X_3$	$X_{13} : X_{12}, X_9, X_{11}, X_7$
$X_2 : X_0, X_1$	$X_8 : X_6, X_5, X_3, X_7$	$X_{14} : X_{12}, X_{13}, X_{11}, X_7$
$X_3 : X_2, X_1$	$X_9 : X_8, X_5, X_3, X_7$	$X_{15} : X_{14}, X_{13}, X_{11}, X_7$
$X_4 : X_2, X_1, X_3$	$X_{10} : X_8, X_9, X_3, X_7$	$X_{16} : X_{14}, X_{13}, X_{11}, X_7, X_{15}$
$X_5 : X_4, X_1, X_3$	$X_{11} : X_{10}, X_9, X_3, X_7$	$X_{17} : X_{16}, X_{13}, X_{11}, X_7, X_{15}$
$X_6 : X_4, X_5, X_3$	$X_{12} : X_{10}, X_9, X_{11}, X_7$	$X_{18} : X_{16}, X_{17}, X_{11}, X_7, X_{15}$

Можно показать, что вычисления всегда завершаются во втором цикле, т.е. при $n < \mu + 2\lambda$. Более подробно этот алгоритм описан в [39].

Функция линейки используется в задаче о Ханойских башнях. Перенумеруем n дисков от 0 до $n-1$. При каждом перемещении k, где k принимает значения от 1 до 2^n-1, циклически перемещаем диск $\text{ntz}(k)$ на минимальную разрешенную дистанцию вправо.

Функция линейки используется также при генерации отраженного бинарного кода Грея (см. раздел 13.1). Начиная с произвольного n-битового слова на каждом шаге k (где k принимает значения от 1 до 2^n-1), изменяем бит $\text{ntz}(k)$.

ГЛАВА 6

ПОИСК В СЛОВЕ

6.1. Поиск первого нулевого байта

Необходимость в отдельной функции поиска нулевого байта возникает главным образом из-за способа представления символьных строк в языке C. Строки не содержат явно указанной длины; вместо этого в конце строки помещается нулевой байт. В C для вычисления длины строки используется функция `strlen`. Эта функция сканирует строку слева направо до тех пор, пока не находит нулевой байт, и возвращает количество просканированных байтов без учета нулевого.

Быстрая реализация функции `strlen` может загружать в регистр целое слово за раз и искать в нем нулевой байт. На компьютерах, где первым хранится старший байт слова, функция должна возвращать смещение первого нулевого байта слева. Удобно использовать значения от 0 до 3, которые соответствуют номеру нулевого байта в слове; если нулевого байта в слове нет, возвращается значение 4. Это значение по ходу поиска добавляется к длине строки. На компьютерах, где первым хранится младший байт, соответствующая проверка должна возвращать номер первого нулевого байта справа, поскольку порядок байтов при загрузке слова в регистр на таких компьютерах изменяется на обратный. Определим функции $\text{zbytel}(x)$ и $\text{zbyter}(x)$ (“00” обозначает нулевой байт, “nn” — ненулевой байт, байт “xx” может быть как нулевым, так и ненулевым).

$$\text{zbytel}(x)=\begin{cases}0, & x=\text{00xxxxxx},\\ 1, & x=\text{nn00xxxx},\\ 2, & x=\text{nnnn00xx},\\ 3, & x=\text{nnnnnn00},\\ 4, & x=\text{nnnnnnnn}.\end{cases}\qquad \text{zbyter}(x)=\begin{cases}0, & x=\text{xxxxxx00},\\ 1, & x=\text{xxxx00nn},\\ 2, & x=\text{xx00nnnn},\\ 3, & x=\text{00nnnnnn},\\ 4, & x=\text{nnnnnnnn}.\end{cases}$$

В листинге 6.1 приведен код функции $\text{zbytel}(x)$, которая ищет *первый нулевой байта слева*. Проверка выполняется последовательно, слева направо, после чего возвращается номер первого слева нулевого байта (если таковой обнаружен).

Листинг 6.1. Поиск первого слева нулевого байта с использованием ветвления

```
int zbytel (unsigned x)
{
    if              ((x >> 24) == 0) return 0;
    else if ((x & 0x00FF0000) == 0) return 1;
    else if ((x & 0x0000FF00) == 0) return 2;
    else if ((x & 0x000000FF) == 0) return 3;
    else return 4;
}
```

Для вычисления функции $\text{zbytel}(x)$ требуется выполнить от 2 до 11 базовых RISC-команд (11 команд в случае, если в слове нет нулевых байтов, что является наиболее распространенной ситуацией для функции `strlen`). Код для функции $\text{zbyter}(x)$ практически аналогичен коду $\text{zbytel}(x)$.

В листинге 6.2 функция zbytel(x) вычисляется без использования команд ветвления. Идея заключается в том, чтобы преобразовать каждый нулевой байт в 0x80, а каждый ненулевой — в нулевой, после чего достаточно воспользоваться функцией подсчета ведущих нулевых битов. Если в компьютере есть команда для вычисления количества ведущих нулевых битов и команда *или-не,* то для поиска нулевого байта потребуется восемь команд. Похожий алгоритм описан в [42].

Листинг 6.2. Поиск первого слева нулевого байта без использования ветвления

```
int zbytel (unsigned x)
{
    unsigned y;
    int n;
                            // Исходный байт: 00 80 другие
    y = (X & OX7F7F7F7F) + 0x7F7F7F7F;  // 7F 7F 1xxxxxxx
    y = ~(y | X | 0x7F7F7F7F);          // 80 00 00000000
    n = nlz(y) >> 3;            // n = 0 ... 4; 4 если в x
    return n;                   // нет нулевого байта
}
```

Позицию первого нулевого байта справа можно получить, разделив нацело количество завершающих нулевых битов в у на 8 (с отбрасыванием дробной части). Если воспользоваться выражением для вычисления завершающих нулевых битов с использованием команды nlz (см. раздел 5.4), то в приведенном выше коде строку, в которой вычисляется п, можно заменить строкой

```
n = (32 - nlz(~y & (y - 1))) >> 3;
```

Таким образом, если в компьютере имеются команды *или-не* и *и-не*, решение поставленной задачи будет получено за 12 команд.

Следует отметить, что на PowerPC в большинстве случаев процедура поиска первого справа нулевого байта не нужна; вместо нее можно воспользоваться командой *загрузки слова с обратным порядком байтов* lwbrx.

Процедура из листинга 6.2 крайне ценна для 64-разрядных компьютеров. Дело в том, что на 64-разрядном компьютере эта процедура (с необходимыми вполне очевидными изменениями) выполняется примерно за то же количество команд (семь или десять — в зависимости от способа генерации констант), что и на 32-разрядной. Процедура с использованием команд ветвления (листинг 6.1) в худшем случае потребует для выполнения 23 команды.

Если требуется только проверить, есть ли в данном слове нулевой байт, то после второго присвоения у можно добавить команду *ветвления при нулевом значении* (или, напротив, *ненулевом)*.

Если команда nlz отсутствует, поиск первого нулевого байта усложняется. В листинге 6.3 показан один из вариантов поиска нулевого байта в слове без использования функции nlz (приведена только исполняемая часть кода).

Листинг 6.3. Поиск первого слева нулевого байта без использования функции nlz

```
                        // Исходный байт: 00 80 другие
    y = (x & OX7F7F7F7F) + 0x7F7F7F7F; //7F 7F Ixxxxxxx
    y = ~(y | X | 0x7F7F7F7F);         //80 00 00000000
                                       //Эти шаги отображают:
    if (y == 0} return 4;              //00000000 ==> 4,
    else if (y > OxOOOOFFFF)           //80xxxxxx ==> O,
```

```
    return (y >> 31) * 1;         //0080xxxx ==> 1,
  else                            //000080xx ==> 2,
    return (y >> 15) ^3;          //00000080 ==> 3,
```

Этот код требует выполнения от 10 до 13 базовых RISC-команд (10 команд в случае, когда в слове нет нулевого байта). Таким образом, этот алгоритм, пожалуй, уступает коду из листинга 6.1, хотя и имеет меньшее количество команд ветвления. К сожалению, масштабирование данного алгоритма для 64-разрядных компьютеров не эффективно.

Рассмотрим еще одну возможность избежать использования функции nlz. Значение y, вычисленное в листинге 6.3, состоит из четырех байтов, каждый из которых равен либо 0x00, либо 0x80. Остаток от деления такого числа на 0x7F представляет собой исходное число, единичные биты которого (их не более четырех) перемещены в четыре крайние справа позиции. Таким образом, остаток от беззнакового деления в диапазоне от 0 до 15 однозначно идентифицирует исходное число, например:

$$\text{remu}(\mathbf{0x80808080}, \mathbf{127}) = 15,$$
$$\text{remu}(\mathbf{0x80000000127}) = 8,$$
$$\text{remu}(\mathbf{0x00008080}, \mathbf{127}) = 3.$$

Это значение можно использовать при поиске нулевого байта в качестве индекса в таблице размером 16 байт. Таким образом, часть кода, которая начинается со строки if (y == 0), можно заменить следующим (здесь y — беззнаковое):

```
static char table[16] = {4, 3, 2, 2, 1, 1, 1, 1,
                         0, 0, 0, 0, 0, 0, 0, 0};
return table[y%127];
```

Вместо 127 можно воспользоваться числом 31, но, разумеется, с другим массивом table.

На практике методы, использующие остаток от деления на 127 или 31, не применяются, так как получение остатка от деления занимает 20 и более тактов даже при непосредственной аппаратной реализации этой команды. Тем не менее код из листинга 6.3 можно усовершенствовать, заменив часть кода, начинающуюся со строки if (y == 0), одной из двух следующих строк:

```
return table[hopu(y, 0x02040810) & 15];
return table [y*0x00204081 >> 28];
```

Здесь hopu(a,b) означает старшие 32 бита беззнакового произведения a и b. Во второй строке в соответствии с соглашением, принятым в языках высокого уровня, предполагается, что значение произведения представляет собой младшие 32 бита результата. Этот метод вполне практичен, особенно на компьютерах с быстрым умножением, а также на тех, у которых есть команда *сдвига-и-сложения*, так как в последнем случае умножение на 0x00204081 реализуется как

$$y\left(1+2^7+2^{14}+2^{21}\right)=y\left(1+2^7\right)\left(1+2^{14}\right).$$

При использовании такого 4-тактового умножения общее время вычислений составляет 13 тактов (7 тактов для вычисления y, 4 — для выполнения команд *сдвига-и-сложения*, 2 — для *сдвига вправо* на 28 и индексирования массива table); кроме того, этот код не содержит команд условного перехода.

Этот алгоритм достаточно просто масштабируется для применения на 64-разрядных компьютерах. В алгоритме с остатком от деления используем инструкцию

```
return table[y%511];
```

Здесь table содержит 256 элементов со значениями 8, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, ... (т.е. table [i] равно количеству завершающих нулевых битов в i).

В мультипликативных методах используется одна из двух инструкций:

```
return table[hopu(y, 0x0204081020408100) & 255];
return table[y*0x0002040810204081>>56];
```

Здесь table имеет размер 256 и значения 8, 7, 6, 6, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 3, ...

Умножение на 0x2040810204081 можно выполнить за 13 тактов следующим образом:

$$t_1 \leftarrow y\left(1+2^7\right),$$
$$t_2 \leftarrow t_1\left(1+2^{14}\right),$$
$$t_3 \leftarrow t_2\left(1+2^{28}\right).$$

Очевидно, все варианты поиска с использованием таблиц легко реализуют поиск первого справа нулевого байта путем простого изменения данных в таблице.

Если на 32-разрядном компьютере нет отдельной команды *или-не*, то вместо команды *не* во втором присвоении у в листинге 6.3 можно использовать один из трех описанных выше операторов return с массивом table [i] = {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 4}. Эта схема не работает для 64-битового компьютера.

Рассмотрим еще одну интересную вариацию процедуры из листинга 6.2, которая не использует функцию nlz. Пусть a, b, c и d — однобитовые переменные, означающие предикаты "первый байт x ненулевой", "второй байт x ненулевой" и т.д. Тогда

$$\text{zbytel}(x) = a + ab + abc + abcd.$$

Здесь умножение реализуется при помощи команды *и*, что дает процедуру, показанную в листинге 6.4 (приведена только выполняемая часть кода).

Листинг 6.4. Поиск первого слева нулевого байта вычислением полинома

```
y = (x & 0X7F7F7F7F) + 0x7F7F7F7F;
y = y | x;                    // 1 в ненулевых байтах

t1 =  y >> 31;                // t1 = a
t2 = (y >> 23) & t1;          // t2 = ab
t3 = (y >> 15) & t2;          // t3 = abc
t4 = (y >>  7) & t3;          // t4 = abcd
return t1 + t2 + t3 + t4;
```

Всего требуется выполнить 15 базовых RISC-команд. Этот алгоритм работает не очень быстро, зато часть вычислений можно распараллелить. На суперскалярном компьютере, которая способна выполнять параллельно до трех независимых арифметических команд, выполнение алгоритма займет 10 тактов.

Внесение небольших изменений позволяет легко найти первый справа нулевой байт:

$$\text{zbyter}(x) = abcd + bcd + cd + d.$$

(Здесь для вычислений потребуется на одну команду *и* больше, чем в коде из листинга 6.4)

Некоторые обобщения

Функции zbytel и zbyter могут использоваться при поиске байта, содержащего некоторое заданное число. Для этого над аргументом x и словом, в каждом байте которого содержится нужное значение, производится операция *исключающего или*, и в получив-

шемся слове осуществляется поиск нужного байта. Например, чтобы найти ASCII-пробел (0x20) в слове *x,* необходимо найти нулевой байт в слове $x \oplus \mathbf{0x20202020}$.

Аналогично, для поиска одинаковых байтов в словах *x* и *y* выполняется поиск нулевого байта в слове $x \Phi \boldsymbol{y}$.

Код из листинга 6.2 и его модификации с небольшими изменениями можно использовать и для поиска, не связанного с границами байта. Например, пусть требуется найти нулевое значение (если таковое есть) в первых четырех битах, следующих за ними 12 битах или в последних 16. Для этого можно использовать код из листинга 6.2 с маской 0x77FF7FFF [51] (если длина поля равна 1, то в соответствующем разряде маски устанавливается 0).

Поиск в заданном диапазоне значений

Код из листинга 6.2 можно легко модифицировать для поиска байта, значение которого находится в диапазоне от 0 до некоторого заданного значения, меньшего 128. В приведенном ниже фрагменте вычисляется индекс первого слева байта, значение которого находится в диапазоне от 0 до 9.

```
y = (X & OX7F7F7F7F) + 0x76767676;
y = y | x;
y = y | 0x7F7F7F7F;          // Байты >  9 = OxFF
y = -y;                      // Байты >  9 = 0x00
                             // Байты <= 9 = 0x80
n = nlz(y) >> 3;
```

Рассмотрим более общий случай. Предположим, что в слове требуется найти первый слева байт, значение которого находится в диапазоне от a до *b*, причем разность верхней и нижней границ этого диапазона меньше 128. Например, в ASCII прописные буквы имеют значения от 0x41 до 0x5A. Чтобы найти первую прописную букву в слове, вычтем из него число 0x41414141 так, чтобы при этом не возникло распространения заемов за пределы байта. После этого поиск байта, содержащего значение из диапазона от 0 до 0x19 (разность 0x5A−0x41), выполняется так же, как и в приведенном выше коде. Используя формулы для вычитания из раздела 2.17 с очевидными упрощениями для $y = 0x41414141$, получим

```
d = (x | 0x80808080) - 0x41414141;
d = ~((x | OX7F7F7F7F) ^ d);
y = (d & OX7F7F7F7F) + 0x66666666;
y = y I d;
y = y | 0x7F7F7F7F; // Байты, не равные 41-5A, становятся FF
y = -y;             // Байты, не равные 41-5A, становятся 00
                    // Байты,    равные 41-5A, становятся 80
n = nlz(y) >> 3;
```

Для некоторых диапазонов значений возможен более простой код. Например, чтобы найти первый байт, содержащий значение в диапазоне от 0x30 до 0x39 (десятичная цифра в кодировке ASCII), необходимо выполнить операцию *исключающего или* с исходным словом и числом 0x30303030, а затем воспользоваться приведенным выше кодом для поиска байта со значением из диапазона от 0 до 9 (это упрощение применимо в тех случаях, когда *n* старших битов верхней и нижней границ диапазона совпадают и нижняя граница заканчивается 8 - и нулевыми битами).

Эти методы могут быть адаптированы для работы с большими диапазонами без дополнительных команд. Например, для поиска индекса первого слева байта, значение ко-

торого находится в диапазоне от 0 до 137 (0x89), строку программы поиска байта со значением от 0 до 9 y - y | x следует заменить строкой y = y & x.

Аналогично, заменив строку y - y | d в программе поиска первого слева байта со значением от 0x41 до 0x5A строкой y - y & d, получим код для поиска первого слева байта, содержащего значение в диапазоне от 0x41 до 0xDA.

6.2. Поиск строки единичных битов заданной длины

Задача заключается в поиске в слове, находящемся в регистре, первой строки из единичных битов, длина которой не менее *n*. *В* результате поиска возвращается позиция первого бита строки; если такой строки в слове не обнаружено, выдается некоторое специальное значение. Вариантами этой задачи являются задача ответа на вопрос о существовании данной строки (возвращается значение *да* или *нет)* и задача поиска строки, состоящей ровно из *n* единичных битов. Задача находит применение в программах распределения дискового пространства, в частности при дефрагментации диска (перемещения данных на диске таким образом, чтобы в результате все блоки, в которых хранится некоторый файл, располагались непрерывно, друг за другом). Эта задача была предложена мне Альбертом Чангом (Albert Chang), который указал, что при ее решении можно использовать функцию определения количества ведущих нулевых битов.

Далее предполагается, что компьютер имеет команду (либо подпрограмму) вычисления функции nlz, определяющей количество ведущих нулевых битов.

Первым в голову приходит простейший алгоритм: вычислить количество ведущих нулевых битов и выполнить сдвиг влево на полученное число. Затем вычисляется количество ведущих единичных битов (инвертируя слово и подсчитывая количество ведущих нулевых битов). Если количество нулевых битов в этой группе не меньше заданной длины строки, значит, искомая группа найдена. Если количество нулевых битов меньше требуемого, выполняется сдвиг слова влево на количество нулевых битов и все действия повторяются сначала. Соответствующий код приведен ниже. Если найдены *n* идущих подряд единичных битов, возвращается число от 0 до 31, указывающее позицию первого бита первой такой строки. В противном случае возвращается значение 32, указывающее на то, что соответствующая строка не найдена.

```
int ffstr1(unsigned x, int n)
{
    int k, p;

    p = 0;             // Инициализация возвращаемой позиции
    while (x != 0)
    {
        k = nlz(x);    // Удаление ведущих нулевых битов
        x = x << k;    //  (если они есть)
        P = P + k;
        k = nlz(~x);   // Подсчет группы единиц
        if (k >= n)    // Если единиц достаточно,
            return p;  // возвращается найденная позиция
        x = x << k;    // Если единиц не достаточно,
        p = p + k;     // пропускаем их
    }
    return 32;
}
```

Алгоритм вполне применим, если цикл выполняется всего несколько раз, например когда в слове x есть относительно большие группы нулевых и единичных битов. Это условие вполне обычно для приложений, выполняющих распределение дискового пространства. Однако в худшем случае выполнение этого алгоритма занимает довольно много времени: например, при x=0x55555555 и $n \geq 2$ выполняется около 178 команд из полного множества RISC-команд.

Алгоритм, который лучше справляется с наихудшим случаем, основан на последовательности команд *сдвига влево* и *и*. Чтобы понять, как он работает, рассмотрим поиск строки из восьми или более последовательных единичных битов в 32-битовом слове *x*. Ниже показано, как можно выполнить этот поиск.

$$x \leftarrow x \,\&\, (x \ll 1)$$
$$x \leftarrow x \,\&\, (x \ll 2)$$
$$x \leftarrow x \,\&\, (x \ll 4)$$

После первого присвоения единичные биты в *x* указывают начальные положения строк единичных битов с длиной 2. После второго присвоения единичные биты в *x* указывают начальные положения строк с длиной 4 (строк длиной 2, следующих за другими строками длиной 2). После третьего присвоения единичные биты в *x* указывают начальные позиции строк длиной 8. Применение функции nlz к получившемуся слову возвращает положение первого бита строки, длина которой не меньше 8. Если такой строки нет, возвращается 32.

Для разработки алгоритма, который бы работал с любыми длинами *n* от 1 до 32, следует взглянуть на задачу несколько иначе. Прежде всего заметим, что три приведенные выше присвоения можно выполнять в любом порядке. Обратный порядок может быть даже более удобным. Для иллюстрации общего метода рассмотрим случай $n = 10$.

$$x_1 \leftarrow x \,\&\, (x \ll 5)$$
$$x_2 \leftarrow x_1 \,\&\, (x_1 \ll 2)$$
$$x_3 \leftarrow x_2 \,\&\, (x_2 \ll 1)$$
$$x_4 \leftarrow x_3 \,\&\, (x_3 \ll 1)$$

Первая инструкция выполняет сдвиг на $n/2$. После этого задача сводится к поиску строки из пяти последовательных единичных битов в x_1. Это может быть сделано путем сдвига влево на $\lfloor 5/2 \rfloor = 2$ бита и выполнением команды *и*, после чего выполняется поиск строки из трех единичных битов ($3=5-2$). Последние два оператора определяют начальные позиции строк длиной 3 в слове x_2. Сумма величин всех сдвигов всегда равна $n-1$. Соответствующий код показан в листинге 6.5. Время работы кода при n от 1 до 32 составляет от 3 до 36 команд из полного множества RISC-команд.

Если n не очень велико, имеет смысл развернуть цикл (на 32-разрядном компьютере достаточно повторить тело цикла пять раз) и опустить проверку n > 1. Код с развернутым циклом не содержит условных переходов и выполняется за 20 команд (последнее присвоение n можно опустить). Тем не менее для малых значений n лучше использовать код с циклом. В случае малых значений n поиск по алгоритму с развернутым циклом выполняется медленнее, потому что как только переменная n становится равной 1, ее значение больше не изменяется и, таким образом, все последующие действия не изменят значений ни x , ни n. В смысле количества выполняемых инструкций развернутый цикл работает быстрее обычного цикла при n > 5.

Листинг 6.5. Поиск первой строки из я единичных битов с помощью команд *сдвига* и *и*

```
int ffstr1(unsigned x, int n)
{
    int s;

    while (n > 1) {
        s = n >> 1;
        x = x & (x << s);
        n = n - s;
    }
    return nlz(x);
}
```

Поиск строки, содержащей точно *n* единичных битов, требует на шесть команд больше (на четыре, если есть команда *и-не)*. После выполнения всех действий листинга 6.5 единичные биты в x находятся в позициях, где в исходном слове начинается строка из *n* или более единичных битов. Следовательно, после подстановки вычисленного значения x выражение

$$x \,\&\, \neg\left(x \overset{u}{\gg} 1\right) \&\, \neg\left(x \ll 1\right)$$

содержит единичный бит в той позиции, в которой в x находится изолированный единичный бит, т.е. с этой позиции в исходном слове x начинается строка ровно из *n* единичных битов.

Данный алгоритм можно легко адаптировать и для поиска строк длиной и, которые начинаются в определенных позициях. Например, чтобы найти строку, первый бит которой находится на границе байта, необходимо применить команду *и* к конечному значению x и числу 0x80808080.

Этот алгоритм можно использовать и для поиска строк из нулевых битов. При этом либо вначале инвертируются все биты x и затем выполняются все обычные вычисления, либо команды *и* заменяются командами *или,* а x инвертируется непосредственно перед вызовом функции nlz. Ниже показан алгоритм поиска первого (самого левого) нулевого байта (точное определение этой задачи приведено в разделе 6.1).

$$\begin{aligned} &x \leftarrow x \mid (x \ll 4) \\ &x \leftarrow x \mid (x \ll 2) \\ &x \leftarrow x \mid (x \ll 1) \\ &x \leftarrow \text{0x7F7F7F7F} \mid x \\ &p \leftarrow \text{nlz}(\neg x) \overset{u}{\gg} 3 \end{aligned}$$

Всего при этом требуется выполнение 12 команд из полного набора RISC-команд (т.е. этот алгоритм оказывается менее эффективным, чем алгоритм из листинга 6.2, в котором выполняется восемь команд).

ГЛАВА 7

ПЕРЕСТАНОВКА БИТОВ И БАЙТОВ

7.1. Реверс битов и байтов

Под операцией реверса битов подразумевается отражение содержимого регистра относительно середины слова, т.е. размещение битов в слове в обратном порядке:

$$\text{rev}(\mathbf{0x01234567}) = \mathbf{0xE6A2C480}$$
$$\text{rev}(\mathbf{00000001001000110100010101100111}) = 11100110101000101100010010000000$$

Под операцией реверса байтов подразумевается аналогичное отображение четырех байтов регистра. Операция реверса байтов необходима при преобразовании данных между форматом, когда первым идет младший байт (little-endian), который используется, например, DEC и Intel, и форматом, когда первым идет старший байт (big-endian), использующийся большинством других производителей.

Ниже показан эффективный метод реверса битов в слове [3]: в первой строке меняются местами соседние биты, во второй — соседние 2-битовые поля и т.д. Все пять операторов присвоения можно выполнять в произвольном порядке.

```
x = (x & 0x55555555) << 1 | (X & OxAAAAAAAA) >> 1;
x = (x & 0x33333333) << 2 | (X & OxCCCCCCCC) >> 2;
x = (x & OxOFOFOFOF) << 4 | (X & OXFOFOFOFO) >> 4;
x = (x & OxOOFFOOFF) << 8 | (x & OxFFOOFFOO) >> 8;
X = (X & OxOOOOFFFF) << 16 | (x & OxFFFFOOOO) >> 16;
```

Для ряда машин можно воспользоваться усовершенствованием, приведенным в листинге 7.1, которое заключается в том, чтобы избежать больших непосредственно задаваемых значений. Этот код требует выполнения 30 базовых RISC-команд и не использует команд ветвления.

Листинг 7.1. Реверс битов

```
unsigned rev(unsigned x)
{
    x = (x & 0x55555555) << 1 | (x >> 1) & 0x55555555;
    x = (x & 0x33333333) << 2 | (x >> 2) & 0X33333333;
    x = (x & OxOFOFOFOF) << 4 | (x >> 4) & OxOFOFOFOF;
    x = (x << 24) | ((x & OxFFOO) << 8) |
        ((x >> 8) & OxFFOO) | (X >> 24);
    return x;
}
```

Последнее присвоение x в этом коде выполняет реверс байта за девять базовых RISC-команд. Если компьютер способен к выполнению циклического сдвига, вычислить x можно за семь команд:

$$x = \left(\left(x \mathbin{\&} \mathbf{0x00FF00FF}\right) \overset{rot}{\gg} 8\right) \Big| \left(\left(x \overset{rot}{\ll} 8\right) \mathbin{\&} \mathbf{OxOOFFOOFF}\right).$$

На PowerPC операцию реверса байта можно реализовать всего за три команды [26]: *циклический сдвиг влево* на 8 бит, за которым следуют две команды `rlwimi` *(rotate left word immediate then mask insert — циклический сдвиг влево непосредственного заданного слова с последующей вставкой маски).*

Обобщенный реверс битов

В [19] предложено следующее обобщение реверса битов:

```
if (k & 1) x = (x & 0x55555555) <<  1 | (x & 0xAAAAAAAA) >> 1;
if (k & 2) x = (x & 0x33333333) <<  2 | (x & 0xCCCCCCCC) >> 2;
if (k & 4) x = (x & 0x0F0F0F0F) <<  4 | (x & 0xF0F0F0F0) >> 4;
if (k & 8) x = (x & 0x00FF00FF) <<  8 | (x & 0xFF00FF00) >> 8;
if (k &16) x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >>16;
```

(Последние две команды *и* можно опустить.) Если k = 31, этот код выполняет реверс всех битов слова. При k = 24 выполняется реверс байтов слова. Если k = 7, выполняется реверс битов в каждом байте, но расположение байтов в слове при этом не изменяется. При k = 16 выполняется перестановка левого и правого полуслов, и т.д. В общем случае бит, находящийся в позиции *m*, перемещается в позицию $m \oplus k$. На аппаратном уровне этот процесс реализуется подобно циклическому сдвигу (пять уровней мультиплексирования, каждым из которых управляет свой бит величины сдвига k).

Современные методы реверса битов

В [25, item 167] приводятся более сложные выражения для реверса 6-, 7- и 8-битовых целых чисел. Несмотря на то что все эти выражения разработаны для 36-битовых машин, выражение для реверса 6-битовых целых чисел работает и на 32-битовых машинах, а выражения для реверса 7- и 8-битовых чисел — на 64-битовых машинах. Приведем эти выражения.

6 бит: $\text{remu}\left(\left(x * \text{0x00082082}\right) \,\&\, \text{0x01122408}, 255\right)$

7 бит: $\text{remu}\left(\left(x * \text{0x40100401}\right) \,\&\, \text{0x442211008}, 255\right)$

8 бит: $\text{remu}\left(\left(x * \text{0x202020202}\right) \,\&\, \text{0x10884422010}, 1023\right)$

В результате получаются "чистые" целые числа, т.е. числа, выровненные по правой границе, неиспользуемые старшие биты которых равны 0.

Во всех трех случаях вместо функции remu можно использовать функции rem или mod, поскольку аргументы функции remu положительны. Функция *остаток от деления* в данных формулах просто суммирует цифры числа в системе счисления по основанию 256 или 1024, подобно тому как в десятичной системе счисления остаток от деления на 9 просто равен сумме цифр числа (если полученное число больше 9, процесс суммирования цифр повторяется). Следовательно, эту функцию можно заменить командами *умножения* и *сдвига вправо*. Например, реверс 6-битовых чисел на 32-битовом компьютере можно выполнить так (умножение выполняется по модулю 2^{32}):

$$t \leftarrow \left(x * \text{0x00082082}\right) \,\&\, \text{0x01122408}$$
$$\left(t * \text{0x01010101}\right) \overset{u}{\gg} 24$$

Применение описанных формул ограничено, так как они включают получение остатка (требующее для выполнения 20 тактов и больше) и/или несколько команд

умножения, а также команды загрузки больших констант. Последняя формула, например, требует выполнения 10 базовых RISC-команд, две из которых — команды *умножения,* что составляет примерно 20 тактов на современных RISC-машинах. С другой стороны, адаптация кода из листинга 7.1 для реверса 6-битовых целых чисел потребует около 15 команд и примерно от 9 до 15 тактов, в зависимости от возможностей конкретной машины по распараллеливанию вычислений на уровне команд. Тем не менее подобные формулы дают очень компактный код. Ниже приведено еще несколько аналогичных алгоритмов для 32-битовой машины. При применении этого метода для 8- и 9-битовых целых чисел на 32-битовой машине в алгоритмах дважды применяются идеи из [25].

Вот как выглядит методика реверса 8-битовых целых чисел:

$$
\begin{aligned}
&s \leftarrow (x * \mathrm{0x02020202}) \,\&\, \mathrm{0x84422010} \\
&t \leftarrow (x * 8) \,\&\, \mathrm{0x00000420} \\
&\mathrm{remu}(s + t, 1023)
\end{aligned}
$$

Здесь функция remu не может быть заменена командами *умножения* и *сдвига.* (Чтобы понять, почему такая замена в данном случае оказывается некорректной, выполните все описанные действия и посмотрите, какие при этом получаются битовые комбинации.)

Вот еще один метод реверса 8-битового целого числа, который интересен тем, что может быть несколько упрощен:

$$
\begin{aligned}
&s \leftarrow (x * \mathrm{0x00020202}) \,\&\, \mathrm{0x01044010} \\
&t \leftarrow (x * \mathrm{0x00080808}) \,\&\, \mathrm{0x02088020} \\
&\mathrm{remu}(s + t, 4095)
\end{aligned}
$$

Упрощение заключается в том, что второе произведение представляет собой *сдвиг влево* первого произведения, последняя маска может быть получена из второй маски путем единственной команды *сдвига,* а команду *остаток от деления* можно заменить командами *умножения* и *сдвига.* После всех упрощений код требует выполнения 14 базовых RISC-команд, две из которых — команды *умножения.*

$$
\begin{aligned}
&u \leftarrow x * \mathrm{0x00020202} \\
&m \leftarrow \mathrm{0x01044010} \\
&s \leftarrow u \,\&\, m \\
&t \leftarrow (u \ll 2) \,\&\, (m \ll 1) \\
&(\mathrm{0x01001001} * (s + t)) \gg 24
\end{aligned}
$$

Реверс 9-битового целого числа выполняется следующим образом:

$$
\begin{aligned}
&s \leftarrow (x * \mathrm{0x01001001}) \,\&\, \mathrm{0x84108010} \\
&t \leftarrow (x * \mathrm{0x00040040}) \,\&\, \mathrm{0x00841080} \\
&\mathrm{remu}(s+t, 1023)
\end{aligned}
$$

Второго умножения можно избежать, так как оно равно первому произведению, сдвинутому вправо на 6 бит. Последняя маска равна второй маске, сдвинутой вправо на 8 бит. После соответствующих замен код будет требовать выполнения 12 базовых RISC-команд, включая одну команду *умножения* и одну *остатка от деления.* Команда *остаток от деления* должна быть беззнаковой, и ее нельзя заменить командами *умножения* и *сдвига.*

Читатель, изучивший эти необычные методы, может создать аналогичный код и для других операций перестановки битов. Рассмотрим следующий простой (и достаточно ис-

кусственный) пример. Пусть имеется 8-битовая величина, из которой требуется извлечь каждый второй бит, и упаковать четыре полученных бита в крайние правые разряды, т.е. выполнить следующее преобразование:

```
0000 0000 0000 0000 0000 0000 abcd efgh ⇒
0000 0000 0000 0000 0000 0000 0000 bdfh
```

Данное преобразование можно выполнить следующим образом:

$$t \leftarrow (x * \mathbf{0x01010101})\ \&\ \mathbf{0x40100401}$$
$$(t * \mathbf{0x08040201}) \overset{u}{\gg} 27$$

Тем не менее на большинстве компьютеров наиболее практичным оказывается метод, основанный на индексировании таблицы из однобайтовых (или 9-битовых) целых чисел.

Увеличение обращенного целого числа

В алгоритме быстрого преобразования Фурье (БПФ) используется как целая переменная i в цикле, в котором происходит ее увеличение на 1, так и значение rev(») [50]. Простейший способ работы состоит, конечно, в вычислении в каждой итерации увеличенного значения i с последующим вычислением $\mathrm{rev}(i)$. Безусловно, для малых значений i быстрее и практичнее проводить поиск значений $\mathrm{rev}(i)$ в таблице, но что делать при больших i, когда поиск в таблице неприменим в силу непрактичности, а для вычисления rev(i) требуется 29 команд?

Если использовать поиск в таблице невозможно, лучше хранить *i* как в нормальном, так и в реверсном виде, увеличивая оба значения при каждой итерации. При этом возникает вопрос: как наиболее эффективно выполнить приращение целого числа, которое хранится в регистре в реверсном виде? Например, на 4-битовой машине при таком увеличении необходимо последовательно пройти такие значения (в шестнадцатеричной записи):

O, 8, 4, C, 2, A, 6, E, 1, 9, 5, D, 3, B, 7, F.

В алгоритме быстрого преобразования Фурье как *i*, так и реверсное значение имеют одинаковую длину, которая почти наверняка меньше 32, и оба значения в регистрах выровнены по правой границе. Однако мы считаем, что i — 32-битовое целое число. После прибавления 1 к реверсному 32-битовому значению выполняется *сдвиг вправо* определенного количества битов, и затем полученное значение используется в алгоритме БПФ для индексации массива в памяти.

Непосредственный метод приращения реверсного значения состоит в выполнении последовательного сканирования слова слева направо до тех пор, пока не будет обнаружен первый нулевой бит. Этот бит устанавливается равным 1, а все биты, расположенные слева от него (если таковые имеются), сбрасываются в 0. Вот код этого метода:

```
unsigned x, m;

m = 0x80000000;
x = x * m;
if ((int)x >= 0)
{
    do
    {
        m = m >> 1;
        x = x ^m;
```

```
    } while (x < m);
}
```

Здесь происходит выполнение трех базовых RISC-команд, если x начинается с нулевого бита, плюс четыре дополнительные команды для каждой итерации цикла. Поскольку вероятность того, что x начинается с нулевого бита, равна $1/2$, что x начинается с 10 (в бинарной записи) равна $1/4$ и т.д., среднее число выполняемых команд равно

$$
\begin{aligned}
&3\cdot\frac{1}{2}+7\cdot\frac{1}{4}+11\cdot\frac{1}{8}+15\cdot\frac{1}{16}+\ldots=\\
&=4\cdot\frac{1}{2}+8\cdot\frac{1}{4}+12\cdot\frac{1}{8}+16\cdot\frac{1}{16}+\ldots-1=\\
&=4\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\ldots\right)-1=\\
&=7.
\end{aligned}
$$

(Во второй строке мы добавили и вычли 1, причем первая единица представлена в виде суммы $1/2+1/4+1/8+\ldots$. Полученный ряд похож на ряд, уже рассматривавшийся в разделе 5.4.) Однако в худшем случае выполнять приходится достаточно большое количество команд — 131.

Если в компьютере есть команда вычисления *количества ведущих нулевых битов* в слове, то увеличить реверсное значение на 1 можно следующим образом:

сначала вычисляется $s \leftarrow \mathrm{nlz}(\neg x)$,

а затем либо $x \leftarrow x \oplus \left(\mathtt{0x80000000} \overset{u}{\gg} s\right)$,

либо $x \leftarrow \left(\left(x \ll s\right) + \mathtt{0x80000000}\right) \overset{u}{\gg} s$.

В любом случае выполняется пять команд из полного набора RISC-команд, а кроме того, для корректного перехода от 0xFFFFFFFF к 0 требуется, чтобы сдвиги выполнялись по модулю 64. (Из-за этого приведенные формулы не будут работать на компьютерах с процессором Intel x86, поскольку сдвиги в нем выполняются по модулю 32.)

7.2. Перемешивание битов

Другим важным видом перестановки битов в слове является операция "идеального перемешивания" (perfect shuffle), которая используется в криптографии. Существует две разновидности идеального перемешивания, которые называются "внешним" и "внутренним". Обе операции чередуют биты из двух половин слова. В целом процесс аналогичен тасованию колоды из 32 карт, разница лишь в том, какая карта должна быть первой. В результате внешнего идеального перемешивания внешние биты остаются во внешних позициях, а после внутреннего идеального перемешивания бит из позиции 15 перемещается в позицию 31. Если 32-битовое слово равно (здесь каждая буква означает один бит)

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP,
```

то после выполнения внешнего идеального перемешивания расположение битов в слове окажется следующим:

```
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP,
```

а после внутреннего полного перемешивания:

AaBb CcDd EeFf GgHh IiJj KkLl MmNn OoPp.

Пусть длина слова W представляет собой степень 2. Тогда операцию внешнего идеального перемешивания можно выполнить при помощи базовых RISC-команд за $\log_2(W/2)$ шагов, причем на каждом шаге выполняется перестановка второй и третьей четвертей во все меньших частях слова [19], т.е. 32-битовое слово преобразуется следующим образом:

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP
abcd efgh ABCD EFGH ijkl mnop IJKL MNOP
abcd ABCD efgh EFGH ijkl IJKL mnop MNOP
abAB cdCD efEF ghGH ijIJ klKL mnMN opOP
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP
```

Приведем код, непосредственно реализующий описанный алгоритм и требующий выполнения 42 базовых RISC-команд.

```
X = (x & 0x0000FF00) << 8 | (X >> 8) & 0x0000FF00 | X & 0xFF0000FF;
X = (x & 0x00F000F0) << 4 | (X >> 4) & 0x00F000F0 | x & 0xF00FF00F;
X = (X & 0x0C0C0C0C) << 2 | (x >> 2) & 0x0C0C0C0C | X & 0xC3C3C3C3;
x = (x & 0x22222222) << 1 | (x >> 1) & 0x22222222 | x & 0x99999999;
```

Этот код можно сократить до 30 команд (хотя при этом время выполнения возрастет с 17 до 21 такта на машинах с неограниченными возможностями распараллеливания вычислений на уровне команд), если использовать для обмена двух полей регистра метод *исключающего или* (который рассматривался в разделе 2.19). Все используемые величины беззнаковые.

```
t = (x ^ (X >> 8)) & 0x0000FF00; x = x ^t * (t <<8);
t = (X ^ (x >> 4)) & 0x00F000F0; X = X ^t ^(t <<4);
t = (X ^ (X >> 2)) & 0x0C0C0C0C; X = X ^t ^(t << 2);
t = (x^ (X >> 1)) & 0x22222222; x = X ^t ^(t <<1);
```

Для выполнения обратной операции достаточно изменить последовательность обменов на обратную.

```
t = (x ^ (x >> 1)) & 0x22222222; x = x ^t ^ (t << 1);
t = (x ^ (x >> 2)) & 0x0C0C0C0C; x = x ^t ^ (t << 2);
t = (X ^ (X >> 4)) & 0x00F000F0; X = X ^t ^ (t.<<4);
t = (X ^ (x >> 8)) & 0X0000FF00; x = x ^t ^ (t << 8);
```

Использование только последних двух шагов обоих приведенных алгоритмов перемешивает биты в пределах байтов; три последних шага приводят к перемешиванию битов в пределах полуслова и т.д. В двух последних строках этого кода выполняется перемешивание битов в каждом отдельном байте; в трех последних строках — в каждом полуслове и т.д. То же замечание справедливо и для операции, обратной перемешиванию, только в этом случае вместо последних шагов рассматриваются первые.

При внутреннем идеальном перемешивании сначала в регистре осуществляется перестановка правого и левого полуслов, а затем выполняется та же последовательность команд, что и при внешнем идеальном перемешивании. Обмен двух полуслов в регистре можно выполнить следующим образом:

```
X = (X >> 16) | (X << 16);
```

(либо с помощью команды *циклического сдвига* на 16 бит). Операция, обратная внутреннему идеальному перемешиванию, выполняется аналогично, но упомянутая строка добавляется в конец кода.

Если изменить описанный выше процесс перемешивания таким образом, чтобы на каждом шаге выполнялась перестановка *первой* и *четвертой* четвертей во все меньших частях слова, то в результате получим реверс слова, образованного внутренним идеальным перемешиванием.

Пожалуй, стоит упомянуть один частный случай, когда левая половина слова (левое полуслово) x состоит из одних нулевых битов; иначе говоря, требуется разместить биты из правой половины слова x так, чтобы они оказались в каждом втором разряде, т.е. преобразовать исходное 32-битовое слово

```
0000 0000 0000 0000 ABCD EFGH IJKL MNOP
```

в слово

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N 0O0P.
```

Внешнее идеальное перемешивание может быть упрощено так, что будет требовать выполнения 22 базовых RISC-команд. Приведенный же ниже код для нашей частной задачи выполняется только за 19 базовых RISC-команд без увеличения времени работы на компьютере с неограниченными возможностями распараллеливания вычислений на уровне команд (12 тактов при любом методе). Для данного кода не требуется установка левой половины слова x равной 0.

```
x = ((x & 0xFF00) << 8) | (x & 0x00FF);
x = ((x << 4) | x) & 0x0F0F0F0F;
x = ((x << 2) | x) & 0x33333333;
x = ((x << 1) | x) & 0x55555555;
```

Аналогично, код обратной операции (частный случай упаковки; см. раздел 7.4) может быть упрощен до 26 или 29 базовых RISC-команд, в зависимости от того, требуется ли выполнение команды *и* для очистки битов в нечетных разрядах или нет. Код, приведенный ниже, позволяет решить задачу обратного идеального перемешивания для частного случая ненулевых четных битов за 18 или 21 базовую RISC-команду, со временем работы на компьютере с неограниченными возможностями распараллеливания вычислений на уровне команд, составляющим 12 или 15 тактов.

```
x = x & 0x55555555; // (При необходимости)
x = ((x >> 1) | x) & 0x33333333;
x = ((x >> 2) | x) & 0x0F0F0F0F;
x = ((x >> 4) | x) & 0x00FF00FF;
x = ((x >> 8) | x) & 0x0000FFFF;
```

7.3. Транспонирование битовой матрицы

При транспонировании матрицы A получается матрица, столбцы которой являются строками матрицы A, а строки — столбцами. В этом разделе рассматриваются методы транспонирования битовой матрицы, элементы которой являются отдельными битами, упакованными по 8 в одном байте, а строки и столбцы начинаются на границах байтов.

Это несложное на первый взгляд преобразование оказывается чрезвычайно громоздким по количеству выполняемых команд.

На большинстве компьютеров загрузка и сохранение отдельных битов выполняется очень медленно, главным образом из-за кода, который должен извлекать и (что еще сложнее) сохранять отдельные биты. Лучше разделить исходную матрицу на подматрицы размером 8x8, загрузить каждую подматрицу 8x8 в регистры, транспонировать каждую из подматриц в регистрах по отдельности и затем сохранить их в соответствующих местах результирующей матрицы. В этом разделе сначала рассматривается задача транспонирования подматрицы размером 8x8.

Как именно хранится матрица — по строкам или столбцам, значения не имеет: в любом случае выполняются одни и те же действия. Предположим для определенности, что матрица хранится построчно и загрузка подматрицы 8x8 в восемь регистров осуществляется восемью командами *загрузить байт,* адресующими столбцы исходной матрицы (т.е. значения адресов, используемые в командах *загрузить байт,* отличаются на величину, кратную ширине исходной матрицы в байтах). После транспонирования подматрица 8x8 сохраняется в столбце результирующей матрицы, т.е. сохранение подматриц выполняется за восемь команд *сохранить байт* в ячейки, адреса которых отличаются на величину, кратную ширине выходной матрицы в байтах (эта величина может отличаться от ширины исходной матрицы, если, конечно, матрицы не являются квадратными). Таким образом, в регистрах aO, al, ..., a7 у нас есть восемь 8-битовых величин, выровненных по правой границе. Требуется вычислить и разместить в регистрах ЬO, Ы, ..., Ь7 восемь 8-битовых величин, выровненных по правой границе, которые затем будут использоваться командами *сохранить байт.* Каким именно должно быть содержимое регистров после вычислений, показано ниже. Каждая цифра и буква здесь представляет один бит. Заметим, что главная диагональ проходит через бит 7 байта 0 (aO) и бит 0 байта 7 (a7). Те, кому это привычнее, могут считать, что, главная диагональ проходит через бит 0 байта 0 и бит 7 байта 7.

```
aO = 0123 4567          bO = 08до wEMU
al = 89ab cdef          Ы  = 19hp xFNV
a2 • ghij klmn          Ь2 = 2aiq yGOW
a3 = opqr stuv   ==>    b3 = 3bjr ZHPX
a4 = wxyz ABCD          b4 = 4cks AIQY
a5 = EFGH IJKL          b5 = 5dlt BJRZ
a6 = MNOP QRST          b6 = 6emu CKS$
a7 = UVWX YZ$.          b7 = 7fnv DLT.
```

Непосредственное решение задачи транспонирования матрицы состоит в выборе и размещении каждого бита результата, как показано ниже. Умножения и деления представляют сдвиги соответственно влево и вправо.

```
ЬO=(aO &128)     | (al &128)/2  | (a2 &128)/4  | (a3 &128)/8 |
   (a4 &128)/16  | (a5 &128)/32 | (aб &128)/64 | (a7     )/128;
b1= (aO & 64) *2 | (al & 64)    | (a2 & 64) /2 | (a3 & 64) /4 |
   (a4 & 64) /8  | (a5 & 64) /16| (a6 & 64) /32| (a7 & 64)/64;
Ь2=(aO & 32) *4  | (al & 32) *2 | (a2 & 32)    | (a3 & 32) /2 |
   (a4 & 32) /4  | (a5 & 32)/8  | (a6 & 32) /16| (a7 & 32) /32;
Ь3=(aO & 16) *8  | (al & 16) *4 | (a2 & 16) *2 | (a3 & 16)    |
   (a4 & 16) /2  | (a5 & 16) /4 | (a6 & 16) /8 | (a7 & 16) /16;
Ь4=(aO &  8)*16  | (al &  8)*8  | (a2 &  8) *4 | (a3 & 8)*2   |
   (a4 &  8)     | (a5 &  8) /2 | (a6 &  8) /4 | <a7 &  8)/8;
b5= (aO &  4)*32 | (al &  4)*16 | (a2 &  4)*8  | (a3 & 4)*4   |
   (a4 &  4)*2   | (a5 &  4)    | (a6 &  4) /2 | <a7 &  4) /4;
```

```
Ь6=(aO & 2)*64  | (a1 & 2)*32  | (a2 & 2)*16 | (a3 & 2)*8  |
   (a4 & 2)*4   | (a5 & 2)*2   | (aб & 2)    | (a7 & 2)/2;
Ь7=(aO     )*128| (al & 1)*64  | (a2 & 1)*32 | (a3 & 1)*16 |
   (a4 & 1)*8   | (a5 & 1)*4   | (aб & 1)*2  | (a7 & 1);
```

На большинстве компьютеров этот код выполняется при помощи 174 команд (62 команды *и*, 56 команд *сдвига* и 56 команд *или*). Команды *или* можно заменить командами *сложения*. На PowerPC транспонирование подматрицы может быть выполнено всего за 63 команды (семь команд *пересылки регистров* и 56 команд *циклического сдвига влево на непосредственно указанное количество битов с последующей вставкой маски)*. Здесь не учтены ни команды *загрузки* и *сохранения байтов,* ни код получения адресов загружаемых байтов.

Хотя рассмотренный выше код и не кажется очень громоздким для такой сложной задачи, ниже описан другой метод, который превосходит рассмотренный более чем в два раза при работе на RISC-компьютере с базовым набором команд.

На первом шаге битовая матрица 8x8 интерпретируется как 16 битовых матриц размером 2x2 и выполняется транспонирование каждой из них. На втором шаге матрица интерпретируется как четыре подматрицы размером 2x2 (каждый элемент такой подматрицы является матрицей размером 2x2) и выполняется транспонирование всех четырех подматриц 2x2. На последнем шаге матрица интерпретируется как матрица 2x2, элементами которой являются матрицы размером 4x4, и выполняется транспонирование этой матрицы 2x2. Эти преобразования проиллюстрированы ниже.

```
0123 4567       082a 4c6e       08go 4cks       08go wEMU
89ab cdef       193b 5d7f       19hp 5dlt       19hp xFNV
ghij klmn       goiq ksmu       2aiq 6emu       2aiq yGOW
opqr stuv  =>   hpjr ltnv  =>   3bjr 7fnv  =>   3bjr zHPX
wxyz ABCD       wEyG AICK       wEMU AIQY       4cks AIQY
EFGH IJKL       xFzH BJDL       xFNV BJRZ       5dlt BJRZ
MNOP QRST       MUOW QYS$       yGOW CKS$       6emu CKS$
UVWX YZ$.       NVPX RZT.       zHPX DLT.       7fnv DLT.
```

По сравнению с алгоритмом, в котором на каждом шаге приходится выполнять действия над каждым из восьми битов в восьми регистрах, этот метод оказывается более эффективным. Сначала байты упаковываются по четыре в регистр, затем выполняется перестановка битов в этих регистрах, за которой следует распаковка. Код данной процедуры представлен в листинге 7.2. Параметр А представляет собой адрес первого байта подматрицы 8x8 исходной матрицы (размер которой $8m\times8n$ бит). Аналогично, параметр в является адресом первого байта 8×8-подматрицы результирующей матрицы (размером $8n\times8m$ бит). (Таким образом, размер исходной матрицы в байтах — $8m\times n$, результирующей — $8n\times m$).

Листинг 7.2. Транспонирование матрицы размером 8x8 бит

```
void transpose8(unsigned char A[8], int m, int n,
                unsigned char B[8])
{
    unsigned x, y, t;
    // Загружаем массив и упаковываем его в x и y
    x = (A[0]«24)     | (A[m]<<16)   | (A[2*m]<<8) | A[3*m];
    y = (A[4*m]<<24) | (A[5*m]<<16) | (A[6*m]<<8) | A[7*m];

    t = (X * (X >> 7)) & OxOOAAOOAA; X = X ^ t ^ (t « 7);
    t = (y * (y >> 7)) & OxOOAAOOAA; y = y ^ t ^ (t<<7);
```

```
    t = (x ^ (X  >>14)) & 0x0000CCCC; X = X * t '' (t  <<14);
    t = (y ^ (y  >>14)) & OXOOOOCCCC; У  = У * t  ^(t  <<14);

    t = (x & OxFOFOFOFO)  |  ((y >> 4) & 0x0F0F0F0F);
    y = ((x << 4) & OXFOFOFOFO)  |  (y & 0x0F0F0F0F);
    X = t;

    B[0]=x>>24;   B[n]=x>>16;   B[2*n]=x>>8; B[3*n]=x;
    B[4*n]=y>>24; B[5*n]=y>>16; B[6*n]=y>>8; B[7*n]=y;
}
```

Строка

```
t = (x ^ (X  >>  7)) & 0x00AA00AA; X = X ^t  ^(t  <<  7);
```

кажется загадкой. В ней выполняется перестановка битов 1 и 8 (если считать справа), 3 и 10, 5 и 12 и т.д. в слове x; в то же время биты 0, 2, 4 и т.д. остаются на месте. Все перестановки осуществляются с помощью метода перестановки битов (см. раздел 2.19), в котором используются команды *исключающего или*. Слово x до и после первого цикла имеет вид

```
0123 4567 89ab cdef ghij klmn opqr stuv
082a 4c6e 193b 5d7f goiq ksmu hpjr ltnv
```

Чтобы реально сравнить эти методы, для непосредственного метода, описанного ранее, составлена завершенная программа, аналогичная процедуре из листинга 7.2. Обе программы откомпилированы компилятором GNU C для целевого компьютера, подобного RISC-компьютеру с базовым набором команд. Полное количество команд с учетом всех команд *загрузки, сохранения,* кода адресации элементов массивов, пролога и эпилога функций составляет 219 для непосредственного метода и 101 для метода из листинга 7.2. (Пролог и эпилог в данных процедурах были пустыми, если не считать команды возврата из функции.) Вариант кода из листинга 7.2 для 64-битовых RISC-компьютеров с базовым набором команд (когда x и y содержатся в одном регистре) содержит около 85 команд.

Алгоритм из листинга 7.2 осуществляет процесс перестановки битов в блоках растущих размеров. Однако транспонирование матрицы можно реализовать и наоборот — от больших блоков к меньшим. В таком алгоритме на первом шаге матрица размером 8x8 интерпретируется как матрица размером 2x2, элементы которой являются матрицами 4x4, и выполняется транспонирование матрицы 2x2. Затем каждая из четырех матриц 4x4 интерпретируется как матрица 2x2, элементы которой являются матрицами размером 2x2 бит, и выполняется транспонирование всех четырех подматриц 2x2 и т.д. Код в этом случае получается такой же, как и в листинге 7.2, но при этом три группы операторов, выполняющих перестановку битов, выполняются в обратном порядке.

Транспонирование битовой матрицы размером 32x32

Естественно, что при транспонировании матриц больших размеров используется тот же рекурсивный метод, что и при транспонировании матриц 8x8. Таким образом, транспонирование матрицы 32x32 выполняется в пять этапов.

Детали реализации транспонирования матрицы 32x32 существенно отличаются от процедуры, приведенной в листинге 7.2, так как мы полагаем, что матрицу 32x32 невозможно полностью разместить в регистрах общего назначения и поэтому необходимо разработать компактную процедуру, которая индексирует соответствующие слова бито-

вой матрицы для выполнения перестановки битов. Описанный алгоритм работает лучше при движении от больших блоков к меньшим.

На первом этапе исходная матрица рассматривается как четыре матрицы размером 16x16 и преобразуется следующим образом:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} A & C \\ B & D \end{bmatrix}$$

Здесь A обозначает левую половину первых 16 слов исходной матрицы, B — правую половину первых 16 слов и т.д. Очевидно, что такое преобразование может быть выполнено посредством следующих обменов:

правая половина слова 0 с левой половиной слова 16,
правая половина слова 1 с левой половиной слова 17,
...
правая половина слова 15 с левой половиной слова 31.

При реализации этого алгоритма в коде используется индекс k, принимающий значения от 0 до 15. В цикле по k правая половина слова с номером k меняется местами с левой половиной слова $k+16$.

На втором этапе матрица рассматривается как 16 матриц размером 8x8 бит и выполняется следующее преобразование:

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \Rightarrow \begin{bmatrix} A & E & C & G \\ B & F & D & H \\ I & M & K & O \\ J & N & L & P \end{bmatrix}$$

Такое преобразование может быть выполнено при помощи следующих обменов:

битов 0x00FF00FF слова 0 с битами 0xFF00FF00 слова 8,
битов OxOOFFOOFF слова 1 с битами OxFFOOFFOO слова 9 и т.д.

Это означает, что биты 0–7 (младшие восемь битов) слова 0 меняются местами с битами 8-15 слова 8 и т.д. Индексы первого слова в этих перестановках принимают значения k = O, 1, 2, 3, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23. На каждом шаге очередное значение k можно вычислить по формуле $k' = (k+9)\ \&\ \neg 8$. В цикле по k биты слова k меняются местами с битами слова $k+8$.

Аналогично, на третьем этапе меняются местами

биты OxOFOFOFOF слова 0 с битами OxFOFOFOFO слова 4,
биты OxOFOFOFOF слова 1 с битами OxFOFOFOFO слова 5 и т.д.

Индексы битов в первом слове в этих перестановках равны k = O, 1, 2, 3, 8, 9, 10, 11, 16, 17, 18, 19, 24, 25, 26, 27. На каждом шаге очередное значение k вычисляется по формуле $k' - (k+5)\ \&\ \neg 4$. *В* цикле по k биты слова k меняются местами с битами слова $k + 4$.

Все описанные действия в целях компактности представлены в виде отдельной функции на языке С в листинге 7.3 [19]. Внешний цикл выполняет описанные пять этапов; переменная j при этом принимает значения 16, 8, 4, 2 и 1. На каждом этапе формируется маска т, принимающая значения 0x0000FFFF, OxOOFFOOFF, OxOFOFOFOF, 0x33333333 и 0x55555555. (Код вычисления маски компактен и красив: `m = m^(m<<j)`. Обратного ему кода не существует, и в этом основная причина того, что преобразования выполняются от блоков больших размеров к блокам меньших размеров.) Во внутреннем цикле k принимает описанные

выше значения. Здесь выполняется обмен битов *a* [k], определяемых маской т, с битами a [k+j], сдвинутыми вправо на j и выделенными с помощью маски m, что эквивалентно выделению битов a [k+j] с помощью дополнения к маске т. Код, выполняющий обмен, использует методику трех *исключающих или,* описанную в разделе 2.19.

Листинг 7.3. **Компактный код транспонирования битовой матрицы** 32x32

```
void transpose32(unsigned A[32])
{
    int j, k;
    unsigned m, t;
    m = 0x0000FFFF;
    for (j = 16; j != 0; j = j >> 1, m = m ^ (m << j))
    {
        for (k = 0; k < 32; k = (k + j + 1) & ~j)
        {
            t = (A[k] ^ (A[k+j] >> j)) & m;
            A[k] = A[k] ^ t;
            A[k+j] = A[k+j] ^ (t << j);
        }
    }
}
```

Откомпилированный с помощью GNU C для компьютера, аналогичного RISC с базовым набором команд, этот код содержит 31 команду — 20 во внутреннем цикле и 7 во внешнем без учета внутреннего. Следовательно, всего выполняется $4+5(7+16\cdot 20)=1639$ команд. Если при транспонировании матрицы использовать 16 обращений к процедуре транспонирования матрицы 8x8 из листинга 7.2, то в этом случае транспонирование выполняется за $16(101+5)=1696$ команд в предположении, что все 16 вызовов следуют один за другим. Расходы на вызов функции составляют, как показало рассмотрение скомпилированного кода, пять команд. Таким образом, с точки зрения времени выполнения оба метода практически эквивалентны.

С другой стороны, код из листинга 7.3 легко модифицировать для 64-битовой машины и транспонирования битовой матрицы размером 64x64. В этом случае понадобится выполнить примерно $4+6(7+32\cdot 20)=3886$ команд. Транспонирование той же матрицы с вызовом функций транспонирования матрицы 8x8 требует выполнения $64(85+5)=5760$ команд.

В алгоритме все преобразования выполняются без использования дополнительной памяти, следовательно, этот алгоритм может применяться для транспонирования матриц больших размеров, с дополнительными действиями по пересылке подматриц 32x32. Это же можно сделать, поместив результирующую матрицу в другой области памяти и выполняя пересылку подматриц в первой либо последней итерации внешнего цикла.

Около половины команд, выполняемых в функции из листинга 7.3, используются для управления циклом, и функция пять раз загружает и сохраняет всю матрицу. Может быть, имеет смысл развертывание циклов, которое позволит убрать из кода команды управления? Да, имеет, если вас интересует в первую очередь скорость работы программы, если дополнительные расходы памяти не представляют для вас проблемы, если кэш процессора вашего компьютера имеет достаточный размер для хранения больших блоков кода, а главное — при медленном выполнении команд ветвления вашим компьютером. Тело программы будет состоять из 80 (5 • 16) повторных обменов битов при помощи

шести команд. Кроме того, в программе будет 32 команды *загрузки* исходной матрицы и 32 команды *сохранения* результата; итого по меньшей мере 544 команды.

Компилятор GNU C не в состоянии выполнить развертывание по такому большому числу итераций (16 для внутреннего цикла и пять для внешнего). В листинге 7.4 представлен код, в котором это развертывание выполнено вручную. Как видите, данная функция помешает транспонированную матрицу в другой массив, отличный от исходного; однако, если параметрами функции выступает один и тот же массив, по завершении работы программы он будет транспонирован. Количество строк swap в данном коде равно 80. После компиляции GNU C для RISC-компьютера с базовым набором команд код содержит, включая пролог и эпилог, 576 команд, среди которых нет команд ветвления (не считая команды возврата из функции). У целевого компьютера нет команд *сохранить несколько слов* и *загрузить несколько слов,* но он в состоянии сохранять и загружать по два регистра за такт при помощи команд *сохранить двойное слово* и *загрузить двойное слово.*

Листинг 7.4. Линейный код транспонирования матрицы 32x32

```
#define swap(a0, al, j, m) t = (aO ^ (al >>j)) & m; \
                            aO = aO ^ t; \
                            al = al ^ (t << j) ;

void transpose32(unsigned A[32], unsigned B[32])
{
    unsigned m, t;
    unsigned aO, al, a2, a3, a4, a5, a6, a7,
             a8, a9, a10, all, a12, a13, a14, a15,
             a16, a17, a18, a19, a20, a21, a22, a23,
             a24, a25, a26, a27, a28, a29,a3O, a31;

    aO = A[ 0]; al  = A[ 1]; a2  = A[ 2]; a3  = A[ 3];
    a4 = A[ 4]; a5  = A[ 5]; a6  = A[ 6]; a7  = A[ 7];
    ...
    a28 = A[28]; a29 = A[29]; a3O = A[30]; a31 = A[31];

    m = OxOOOOFFFF;
    swap(a0, a16, 16, m)
    swap(a1, a17, 16, m)
    ...
    swap(a15, a31, 16, m)
    m = OxOOFFOOFF;
    swap(a0, a8, 8, m)
    swap(a1, a9, 8, m)
    ...
    ...
    swap(a28, a29, 1, m)
    swap(a30, a31, 1, m)

    B[ 0] = aO; B[ 1] = al; B[ 2] = a2; B[ 3] = a3;
    B[ 4] = a4; B[ 5] = a5; B[ 6] = a6; B[ 7] = a7;
    ...
    B[28] = a28; B[29] = a29; B[30] = a3O; B[31] = a31;
}
```

Производительность кода можно немного увеличить при наличии в компьютере команды *циклического сдвига* (неважно — влево или вправо). Идея состоит в замене всех операций обмена битов из листинга 7.4, каждая из которых выполняется за шесть ко-

манд, более простыми обменами без сдвига, **выполняющимися** за четыре команды каждая (при этом используются имеющиеся макросы, но с опущенными сдвигами).

Сначала выполняется циклический сдвиг вправо на 16 разрядов слов A[16..31] (т.е. $A[k]$ для $16 \le k \le 31$). Затем выполняется обмен местами правых половин слов: A[0] с $A[16]$, A[1] с $A[17]$ и т.д., аналогично коду из листинга 7.4. После этого выполняется циклический сдвиг вправо на восемь разрядов слов A[0..8] и A[24..31] и обмен битами, соответствующими маске 0x00FF00FF, между словами A[0] и $A[8]$, A/\/ и $A[9]$ и т.д. После выполнения пяти этапов транспонирование полностью не завершено. Необходимо выполнить циклический сдвиг слова $A[1]$ на один разряд, слова $A[2]$ — на два разряда и т.д. (всего 31 команда). Код для этого метода не приводится, однако ниже показана последовательность шагов для битовой матрицы 4x4.

```
abcd       abcd       abij       abij       aeim       aeim
efgh  =>   efgh  =>   efmn  =>   nefm  =>   nbfj  =>   bfjn
ijkl       klij       klcd       klcd       kocg       cgko
ranop      opmn       opgh       hopg       hlpd       dhlp
```

Часть программы из листинга 7.4, в которой выполняются перестановки битов, требует выполнения 480 команд (80 перестановок по шесть команд в каждой). В исправленной программе с использованием команд циклического сдвига выполняется 80 перестановок по четыре команды каждая, плюс 80 команд *циклического сдвига* $(16 \cdot 5)$ на первых пяти этапах и завершающая 31 команда *циклического сдвига* — итого 431 команда. Поскольку код пролога и эпилога остается неизменным, использование команд *циклического сдвига* позволяет сэкономить 49 команд.

Существует еще один метод транспонирования битовой матрицы, основанный на трех преобразованиях сдвига [19]. Если имеется матрица размером $n \times n$, то выполняются следующие шаги: 1) циклический сдвиг строки i вправо на i разрядов; 2) циклический сдвиг **столбца** j вверх на $(j+1) \bmod n$ разрядов; 3) циклический сдвиг строки i вправо на $(i+1) \bmod n$ разрядов; 4) отражение матрицы относительно горизонтальной оси, проходящей через ее середину. Проиллюстрируем сказанное на примере матрицы 4x4.

```
abcd       abcd       hlpd       dhlp       aeim
efgh  =>   hefg  =>   kocg  =>   cgko  =>   bfjn
ijkl       klij       nbfj       bfjn       cgko
mnop       nopm       aeim       aeim       dhlp
```

Этот метод не может конкурировать с другими из-за высокой стоимости шага 2. (Чтобы выполнить этот шаг за разумное количество шагов, сначала выполняется циклический сдвиг вверх на л/2 позиции всех столбцов, которые должны быть сдвинуты на $n/2$ разряда или более (это столбцы от л/2-1 до $n - 2$), затем выполняется циклический сдвиг вверх на л/4 позиции и т.д.) Шаги 1 и 3 требуют выполнения всего лишь по л -1 команд каждый, а шаг 4 не требует никаких действий, если сохранять результаты вычислений в соответствующих позициях.

Если битовая матрица 8x8 сохраняется в 64-разрядном слове как обычно (верхняя строка помещается в старшие восемь битов и т.д.), то операция транспонирования такой матрицы эквивалентна трем внешним идеальным перемешиваниям или обратным им

процессам [19]. Если перемешивание реализовано на компьютере в виде отдельной команды, этим можно воспользоваться; однако для RISC-компьютеров с базовым набором команд этот способ не годится.

7.4. Сжатие, или обобщенное извлечение

В языке программирования APL имеется операция, которая называется сжатием (compress) и записывается как B/V, где B — булев вектор, а V — вектор с произвольными элементами той же длины, что и B. Результатом данной операции является вектор, состоящий из тех элементов V, для которых соответствующий бит в B равен 1. Длина результирующего вектора равна количеству единичных битов в B.

Рассмотрим аналогичную операцию с битами в слове. Заданы маска m и слово x; требуется выбрать те биты слова x, которые соответствуют единичным битам маски, и переместить их вправо ("сжать"). Например, если сжимаемое слово

abcd efgh ijkl mnop qrst uvwx yzAB CDEF,

где каждый символ представляет один бит, а маска имеет вид

0000 1111 OOH OOH 1010 1010 0101 0101,

то в результате будет получено слово

0000 0000 0000 0000 efgh klop qsuw zBDF.

Эту операцию можно также назвать ***обобщенным извлечением*** (generalized extract) по аналогии с командой *извлечения,* имевшейся в ряде компьютеров.

Нас интересует код, осуществляющий данную операцию с минимальным временем работы в наихудшем случае, и в качестве первого приближения, которое будет совершенствоваться, возьмем простой цикл из листинга 7.5. Этот код не содержит команд ветвления в теле цикла и в худшем случае требует выполнения 260 команд, включая пролог и эпилог функции.

Листинг 7.5. Операция сжатия на основе простого цикла

```
unsigned compress(unsigned x, unsigned m)
{
    unsigned r, s, b; // Результат, сдвиг, маскируемый бит
    r = 0;
    s = 0;
    do
    {
        b = m & 1;
        r = r I ((x & b) << s);
        s = s + b;
        x = x >> 1;
        rn = m >> 1;
    } while (m != 0);
    return r;
}
```

Улучшить этот код можно путем многократного применения метода "параллельного префикса" (см. раздел 5.2) с операцией *исключающего или* [19] (далее будем обозначать эту операцию как PP-XOR). Основная идея состоит в том, чтобы прежде всего определить биты аргумента x, которые будут перемещены на нечетное количество позиций, и

переместить их (эта задача упрощается, если сначала применить операцию *и к* слову *x* и к маске, очистив биты, которые не имеют значения). Биты маски перемещаются точно таким же образом. Затем определяются биты *x,* перемещаемые на нечетное число, умноженное на 2 (2, 6, 10 и т.д.), после чего вправо перемещаются как эти биты, так и соответствующие биты маски. Далее работаем с битами, перемещаемыми на нечетные числа, умноженные на 4, затем на 8 и 16.

Поскольку этот алгоритм достаточно сложен для **понимания,** рассмотрим его более детально. Предположим, что входными данными для нашего алгоритма являются следующие:

```
x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF,
m = 1000 1000 1110 0000 0000 1111 0101 0101,
    1    1    111
    9    6    333            4444 3 2  1 0
```

Здесь каждая буква в x означает один бит (который может принимать значения 0 или 1). Числа под каждым единичным битом маски указывают, насколько далеко должен перемещаться вправо соответствующий бит x (это значение равно количеству нулевых битов в m справа от рассматриваемого бита). Как упоминалось ранее, сначала стоит сбросить все не имеющие значения биты x, что даст нам число

```
X = aOOO eOOO ijkO 0000 0000 uvwx OzOB ODOF.
```

Сначала нужно определить, какие биты будут перемещены вправо на нечетное количество позиций, и переместить их на одну позицию вправо. Вспомним, что операция PP-XOR дает нам единичные биты в позициях, для которых количество единичных битов в данной позиции и справа от нее нечетно. Нас же интересуют биты, для которых нечетно количество нулевых битов, расположенных строго справа от данной позиции. Найти их можно путем вычисления `mk = ~m<<1` и применением операции PP-XOR к полученному значению. Результат будет следующим:

```
mk = 1110 1110 OOH 1111 1110 0001 0101 0100,
тр = 1010 0101 1110 1010 1010 0000 1100 1100.
```

Заметим, что mk указывает биты т, которые содержат нулевой бит рядом справа, а тр суммирует их справа по модулю 2. Таким образом, тр определяет биты т, которые имеют нечетное количество нулевых битов справа.

Биты, которые должны быть перемещены на одну позицию, находятся в позициях, имеющих нечетное количество нулевых битов справа (указываемые словом тр), и в этих позициях в m находятся единичные биты, т.е. их можно получить с помощью простой операции mv = `mp&m`.

```
mv = 1000 0000 1110 0000 0000 0000 0100 0100
```

Соответствующие биты m могут быть перемещены путем присвоения

```
m = (m * mv) | (mv >> 1);
```

А биты x — с помощью двух присвоений:

```
t = X & mv;
X = (X ^ t) | (t >> 1);
```

(Перемещение битов т выполняется проще в силу того, что здесь все выбранные биты являются единичными.) Операция *исключающего или* сбрасывает биты, которые равны 1 в m и x, а операция *или* устанавливает биты, которые в m и x равны 0. Эти операции могут

быть заменены операциями *вычитания* и *сложения* либо обе быть операциями *исключающего или*. После перемещения битов, выбранных при помощи mv, на одну позицию вправо получаем:

m = 0100 1000 0111 0000 0000 1111 OOH OOH,
x = 0a00 e000 Oijk 0000 0000 uvwx 00zB 00DF.

Теперь необходимо подготовить маску для второй итерации, где будут указаны биты, которые должны быть перемещены вправо на величину, представляющую собой нечетное число, умноженное на 2. Заметим, что величина mk&~mp определяет биты, соседом которых справа в исходной маске m является нулевой бит, и биты, которые в исходной маске имеют справа четное число нулевых битов. Эти свойства применяются совместно, хотя и не каждое в отдельности, к новому значению маски m (т.е. mk указывает *все* позиции в новой маске т, у которых справа расположен нулевой бит, а всего справа находится четное число нулей). Будучи просуммированной справа при помощи операции PP-XOR, эта величина указывает биты, которые перемещаются вправо на количество позиций, равное удвоенному нечетному числу (2, 6, 10 и т.д.). Таким образом, необходимо присвоить это значение переменной mk и выполнить вторую итерацию описанных выше шагов. Новое значение mk имеет вид

mk = 0100 1010 0001 0101 0100 0001 0001 0000.

Завершенная функция С, реализующая рассматриваемую операцию, приведена в листинге 7.6. Она требует выполнения 127 базовых RISC-команд, включая пролог и эпилог. В листинге 7.7 показана последовательность значений, принимаемых различными переменными в ключевых точках вычислений (входные данные те же, что использовались при рассмотрении данного алгоритма). Заметьте, что побочным результатом работы алгоритма является значение маски т, в котором все единичные биты перенесены вправо.

Листинг 7.6. Метод параллельного префикса для операции сжатия

```
unsigned compress (unsigned x, unsigned m)
{
    unsigned mk, mp, mv, t;
    int i;
    x = x & m;                  // Сброс незначащих битов
    mk = ~m << 1;               // Подсчитаем O справа
    for (i = 0; i < 5; i++)
    {
        mp = mk ^ (mk << 1);  // Параллельный префикс
        mp = mp ^ (mp << 2);
        mp = mp ^ (mp << 4);
        mp = mp ^ (mp << 8);
        mp = mp ^ (mp << 16);
        mv = mp & m;            II Перемещаемые биты
        m = m ^ mv | (mv >> (1 << i)); // Сжатие m
        t = x & mv;
        x = x ^t  | (t>> (1 << i));    // Сжатие x
        mk = mk & -mp;
    }
    return x;
}
```

Листинг 7.7. Действия при использовании метода параллельного префикса для операции сжатия

```
                  x  = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
                  m  = 1000 1000 1110 0000 0000 1111 0101 0101
                  x  = a000 e000 ijk0 0000 0000 uvwx 0z0B 0D0F
i = 0,            mk = 1110 1110 0011 1111 1110 0001 0101 0100
После PP,         mp = 1010 0101 1110 1010 1010 0000 1100 1100
                  mv = 1000 0000 1110 0000 0000 0000 0100 0100
                  m  = 0100 1000 0111 0000 0000 1111 0011 0011
                  x  = 0a00 e000 0ijk 0000 0000 uvwx 00zB 00DF
i = 1,            mk = 0100 1010 0001 0101 0100 0001 0001 0000
После PP,         mp = 1100 0110 0000 1100 1100 0000 1111 0000
                  mv = 0100 0000 0000 0000 0000 0000 0011 0000
                  m  = 0001 1000 0111 0000 0000 1111 0000 1111
                  x  = 000a e000 0ijk 0000 0000 uvwx 0000 zBDF
i = 2,            mk = 0000 1000 0001 0001 0000 0001 0000 0000
После PP,         mp = 0000 0111 1111 0000 1111 1111 0000 0000
                  mv = 0000 0000 0111 0000 0000 1111 0000 0000
                  m  = 0001 1000 0000 0111 0000 0000 1111 1111
                  x  = 000a e000 0000 0ijk 0000 0000 uvwx zBDF
i = 3,            mk = 0000 1000 0000 0001 0000 0000 0000 0000
После PP,         mp = 0000 0111 1111 1111 0000 0000 0000 0000
                  mv = 0000 0000 0000 0111 0000 0000 0000 0000
                  m  = 0001 1000 0000 0000 0000 0111 1111 1111
                  x  = 000a e000 0000 0000 0000 0ijk uvwx zBDF
i = 4,            mk = 0000 1000 0000 0000 0000 0000 0000 0000
После PP,         mp = 1111 1000 0000 0000 0000 0000 0000 0000
                  mv = 0001 1000 0000 0000 0000 0000 0000 0000
                  m  = 0000 0000 0000 0000 0001 1111 1111 1111
                  x  = 0000 0000 0000 0000 000a eijk uvwx zBDF
```

Алгоритм из листинга 7.6 на 64-битовой машине с базовым RISC-набором выполняется за 169 команд, в то время как алгоритм из листинга 7.5 в худшем случае требует выполнения 516 команд.

Количество команд, выполнения которых требует алгоритм из листинга 7.6, может быть значительно снижено, если m представляет собой константу. Это может быть в двух ситуациях: 1) когда вызов compress `(x,m)` осуществляется в цикле, в котором значение m не известно заранее, но постоянно в пределах цикла, 2) когда известно значение m и код функции compress генерируется заранее, возможно, компилятором.

Обратите внимание на то, что значение, присвоенное x в цикле в листинге 7.6, используется в этом цикле исключительно в строке, где выполняется присвоение x. Как видно из кода, x зависит только от себя самого и переменной mv. Следовательно, все обращения к x можно удалить, а пять вычисляемых значений mv сохранить в переменных `mv0`, `mv1`, ..., `mv4`. Тогда в ситуации 1 вне цикла, в котором имеется обращение `compress(x,m)`, можно разместить функцию, которая не будет обращаться к x, а будет лишь вычислять значения `mvi`, а в самом цикле можно разместить следующие строки:

```
x = x & m;
t = x & mv0; x = x ^ t | (t >> 1);
t = x & mv1; x = x ^ t | (t >> 2);
t = x & mv2; x = x ^ t | (t >> 4);
t = x & mv3; x = x ^ t | (t >> 8);
t = x & mv4; x = x ^ t | (t >> 16);
```

Таким образом, вычисления в цикле состоят из 21 команды (загрузка констант выносится за пределы цикла), что является значительным улучшением по сравнению со 127 командами, необходимыми в случае полной подпрограммы из листинга 7.6.

В ситуации 2, в которой известно значение т, можно выполнить примерно те же действия, не считая другой возможной оптимизации. Так, может оказаться, что одна из пяти масок равна 0, и в этом случае можно опустить одну из приведенных выше строк. Например, если нет битов, которые должны быть перемещены на нечетное число позиций, маска ml равна 0; если нет битов, которые перемещаются более чем на 15 позиций, равной 0 окажется маска m4.

Так, для

```
m = 0101 0101 0101 0101 0101 0101 0101 0101
```

вычисленные маски равны

```
mvO = 0100 0100 0100 0100 0100 0100 0100 0100
mv1 = OOH  0000 OOH  0000 OOH  0000 OOH  0000
mv2 = 0000 1111 0000 0000 0000 1111 0000 0000
mv3 = 0000 0000 1111 1111 0000 0000 0000 0000
mv4 = 0000 0000 0000 0000 0000 0000 0000 0000
```

Поскольку последняя маска равна 0, в скомпилированном коде операция сжатия выполняется за 17 команд (не считая команд для загрузки масок). Конечно, имеются еще более специализированные случаи, в частности код из раздела 7.2, в котором сжимаются чередующиеся биты, выполняется всего за 13 команд.

Использование команд вставки и извлечения

Если ваш компьютер имеет команду *вставки* (insert), причем предпочтительно с непосредственно задаваемыми значениями в качестве операндов, которые определяют битовое поле в целевом регистре, то она может использоваться для выполнения операции *сжатия* с меньшим числом команд, чем в описанном выше методе. Кроме того, при этом не накладываются ограничения на использование регистров, хранящих маски.

Целевой регистр инициализируется нулевым значением, после чего для каждой непрерывной группы единичных битов в маске m переменная x сдвигается вправо для выравнивания очередного поля и используется команда *вставки,* которая вставляет биты x в соответствующее место целевого регистра. Таким образом операция сжатия выполняется с помощью $2n+1$ команд, где n — количество полей (групп единичных битов) в маске. В наихудшем случае требуется выполнить 33 команды, так как максимальное число полей равно 16 (случай чередования единичных и нулевых битов).

В качестве примера, когда метод с использованием команды вставки дает существенный выигрыш, рассмотрим m=0x0010084A. Сжатие с такой маской требует перемещения битов на 1, 2, 4, 8 и 16 позиций. Следовательно, при использовании метода параллельного префикса требуется выполнение всех 21 команды, в то время как при использовании *вставки* достаточно 11 команд (в маске имеется пять полей). Еще более удачным примером может служить маска m=0x80000000. Здесь происходит перемещение одного бита на 31 позицию, что требует выполнения 21 команды в методе параллельного префикса, только трех команд при использовании вставки и только одной команды *(сдвиг вправо на 31 бит),* если вы не ограничены определенной схемой вычислений.

Можно также воспользоваться командой *извлечения* (extract) для реализации операции сжатия посредством выполнения $3n-2$ команд, где и — количество полей в маске.

Совершенно ясно, что создание оптимального кода, реализующего сжатие для заранее известной маски, представляет собой далеко не тривиальную задачу.

Сжатие влево

Очевидным путем решения этой задачи является реверс аргумента x и маски т, выполнение сжатия вправо и реверс полученного результата. Еще один способ решения состоит в сжатии вправо с последующим сдвигом влево на $\text{pop}(\overline{m})$ позиций. Эти решения наиболее приемлемы, если ваш компьютер имеет команды реверса или подсчета количества единичных битов. Если же таких команд в наборе компьютера нет, можно легко адаптировать алгоритм из листинга 7.6, просто изменив направление всех сдвигов на обратное (за исключением двух сдвигов в выражениях `1<<i`, всего восемь изменений).

7.5. Обобщенные перестановки

При выполнении обобщенной перестановки битов в слове (или где-либо еще) главной проблемой становится представление перестановок. Очень компактно представить перестановку невозможно. Поскольку в 32-битовом слове можно выполнить 32! перестановки, для указания одной из них требуется, как минимум, $\lceil \log_2(32!) \rceil = 118$ битов, или 3 слова и еще 22 бита.

Один интересный способ представления перестановок тесно связан с операцией сжатия, рассматриваемой в разделе 7.4 [19]. Начнем с непосредственного метода, в котором для каждого бита просто указывается позиция его перемещения. Например, для перестановки, которая выполняется путем циклического сдвига влево на четыре бита, бит в позиции 0 (младший бит) переместится в позицию 4, в позиции 1 — в позицию 5, ..., в позиции 31 — в позицию 3. Такая перестановка может быть представлена следующим вектором из 32 пятибитовых индексов:

```
00100
00101
 ...
11111
00000
00001
00010
00011
```

Рассматривая это представление как битовую матрицу, транспонируем ее так, чтобы при этом верхняя строка содержала младшие биты, а результат представим в формате, когда младший бит располагается справа. Таким образом будет получен массив из пяти 32-битовых слов.

```
p[0] = 1010 1010 1010 1010 1010 1010 1010 1010
p[1] = 1100 1100 1100 1100 1100 1100 1100 1100
p[2] = 0000 1111 0000 1111 0000 1111 0000 1111
p[3] = 0000 1111 1111 0000 0000 1111 1111 0000
p[4] = 0000 1111 1111 1111 1111 0000 0000 0000
```

Каждый бит в p [o] представляет собой младший бит позиции, в которую переносится соответствующий бит исходного слова x, биты в p [1] представляют собой следующий значащий бит и т.д. Это очень похоже на то, как маска m в предыдущем разделе ко-

дировалась значениями mv, только в алгоритме сжатия эти значения применялись к обновляемому значению маски, а не к исходному.

Операция сжатия, интересующая нас, должна сжимать влево все биты, помеченные в маске единицами, и вправо — биты, помеченные нулями[1]. Иногда эту операцию называют операцией разделения "агнцев и козлищ" (sheep and goats — SAG) или "обобщенным упорядочением" (general unshuffle). Вычислить эту операцию можно по формуле

```
SAG(x, m) = compress_left(x, m) | compress(x, ~m).
```

Используя SAG в качестве базовой операции, а перестановку p описанную так, как показано выше, биты слова x можно переставить в нужном порядке за 15 шагов.

```
x    = SAG(x,    p[0]);
p[1] = SAG(p[1], p[0]);
p[2] = SAG(p[2], p[0]);
p[3] = SAG(p[3], p[0]);
p[4] = SAG(p[4], p[0]);
x    = SAG(x,    p[1]);
p[2] = SAG(p[2], p[1]);
p[3] = SAG(p[3], p[1]);
p[4] = SAG(p[4], p[1]);
x    = SAG(x,    p[2]);
p[3] = SAG(p[3], p[2]);
p[4] = SAG(p[4], p[2]);
x    = SAG(x,    p[3]);
p[4] = SAG(p[4], p[3]);
x    = SAG(x,    p[4]);
```

Здесь операция SAG используется для выполнения устойчивой двоичной поразрядной сортировки (устойчивость означает сохранение порядка следования равных элементов при сортировке). Массив p используется как 32 пятибитовых ключа для сортировки битов x. На первом шаге все биты x, для которых p [0] = 1, перемещаются в левую половину результирующего слова, а все, для которых p [0] = 0, — в правую. Порядок битов при этом не изменяется (т.е. сортировка устойчива). Затем аналогично сортируются все ключи, которые будут использоваться в следующих шагах алгоритма. Очередная, шестая, строка сортирует x на основе вторых младших битов и т.д.

Подобно сжатию, если некоторая перестановка p используется с рядом слов x, можно значительно сэкономить выполняемые команды путем предварительного вычисления большинства перечисленных шагов алгоритма. Массив перестановки предварительно преобразуется в следующий:

```
p[1] = SAG(p[1], p[0]);
p[2] = SAG(SAG(p[2],p[0]),p[1]);
p[3] = SAG(SAG(SAG(p[3],p[0]),p[1]),p[2]);
p[4] = SAG(SAG(SAG(SAG(p[4],p[0]),p[1]),p[2]),p[3]);
```

После этого каждая перестановка осуществляется с помощью следующих действий:

```
x = SAG(x, p[0]);
x = SAG(x, p[1]);
x = SAG(x, p[2]);
x = SAG(x, p[3]);
x = SAG(x, p[4]);
```

[1] Если используется формат, в котором младший бит находится справа, влево сжимаются биты, помеченные нулями, а помеченные единицами сжимаются вправо.

Более прямой (хотя, пожалуй, менее интересный) способ осуществления обобщенных перестановок битов в слове состоит в представлении перестановки как последовательности 32 пятибитовых индексов. k-й индекс представляет собой номер бита в исходном слове, который будет помещен в k-ю позицию при перестановке. (Список "откуда", в отличие от списка "куда", применяемого при методе GAS.) Эти индексы могут быть упакованы по 6 в 32-битовое слово; таким образом, для хранения всех 32 индексов требуется шесть слов. Аппаратно команда может быть реализована как

bitgather Rt, Rx, Ri,

где Rt — целевой регистр (а также источник), регистр RX содержит переставляемые биты, а в регистре Ri содержится шесть пятибитовых индексов (и два неиспользуемых бита). Ниже приведена операция, выполняемая командой.

$$t \leftarrow (t \ll 6) \mid x_{i_0} x_{i_1} x_{i_2} x_{i_3} x_{i_4} x_{i_5}.$$

Говоря обычным языком, содержимое целевого регистра сдвигается влево на шесть позиций и из слова x выбираются шесть битов и помещаются в шесть освобожденных позиций t. Выбираемые биты определяются шестью пятибитовыми индексами в слове i, берущимися в порядке слева направо. Нумерация битов в индексах может быть как слева направо, так и справа налево.

Для перестановки слова используется последовательность из шести таких команд, все с одинаковыми Rt и Rx, но с разными индексными регистрами. В первом индексном регистре последовательности значащими являются только индексы i_4 и i_5, а биты, выбираемые остальными четырьмя индексами, просто выдвигаются за пределы Rt.

Реализация этой команды, скорее всего, будет позволять индексным значениям повторяться, так что эта команда может использоваться не только для перестановки битов. Она может применяться для многократного повторения любого выбранного бита в целевом регистре. Операции SAG такая универсальность не свойственна.

Не слишком сложно реализовать эту команду как быструю (т.е. выполняемую за один такт). Модуль выбора бита состоит из шести 32:1 мультиплексоров. Если они построены на базе пяти мультиплексоров 2:1, то данная команда может выполняться быстрее, чем 32-битовая команда сложения [45].

Перестановка битов применяется в криптографии, а для компьютерной графики характерна тесно связанная с перестановкой битов операция перестановки подслов (например, перестановка байтов в пределах слова). Оба эти применения работают чаще с 64-битовыми словами, иногда даже со 128-битовыми, чем с 32-битовыми. Методы SAG и *bitgather* легко модифицировать для работы со словами указанной длины.

Для шифрования и дешифрования сообщения с помощью алгоритма стандартного шифрования данных (DES) требуется большое количество отображений, аналогичных перестановкам. Сначала выполняется генерация ключа, которая включает 17 перестановочных отображений. Затем каждый блок сообщения, состоящий из 64 битов, подвергается 34 операциям перестановки. Первая и последняя операции представляют собой 64-битовые перестановки, причем одна из них обратна другой. Кроме того, 16 перестановок с повторениями отображают 32-битовые величины на 48-битовые и еще 16 выполняются в пределах 32-битовых слов [11].

Впрочем, в настоящее время DES вышел из употребления, после того как в 1998 году Electronic Frontier Foundation была доказана его низкая криптостойкость при наличии специального аппаратного обеспечения. Национальный институт стандартов и технологий (National Institute of Standards and Technology — NIST) в качестве временной замены предложил использовать Triple DES — метод шифрования, в котором к каждому

64-битовому блоку DES применяется трижды, каждый раз с другим ключом (таким образом, длина ключа равна 192 битам, включая 24 бита четности). Такой метод требует соответственно в три раза больше операций перестановки битов по сравнению с обычным DES.

Однако предлагаемая "постоянная" замена DES и Triple DES, стандарт Advanced Encryption Standard [1], совсем не использует перестановки на уровне битов. Ближайшая к перестановкам операция, используемая новым стандартом, — простой циклический сдвиг на расстояния, кратные 8. Другие предлагаемые или применяемые методы шифрования используют гораздо меньше перестановок по сравнению с DES.

При сравнении двух рассмотренных методов перестановки следует указать достоинства каждого из них. Метод *bitgather* обладает следующими преимуществами: простота подготовки индексных слов из данных, описывающих перестановку, более простое аппаратное обеспечение и более универсальное отображение. Метод SAG 1) выполняет перестановку за меньшее число команд, 2) использует в команде только два регистра-источника (что может играть существенную роль в ряде RISC-архитектур), 3) легче масштабируется для выполнения перестановки в двойных словах, 4) более эффективно выполняет перестановку подслов.

Пункт 3 подробно рассматривается в [43]. Команда SAG позволяет осуществить перестановку двойного слова путем выполнения двух команд SAG, нескольких базовых RISC-команд и двух полных перестановок единичных слов. Команда *bitgather* позволяет сделать то же путем выполнения *трех* полных перестановок единичных слов и нескольких базовых RISC-команд. Здесь не учитывается предварительная обработка перестановки, состоящая в получении значений, зависящих только от нее; этот подсчет остается читателю в качестве домашнего задания.

Что касается пункта 4, то для перестановки, например, четырех байтов слова при помощи *bitgather* требуется выполнить шесть команд — столько же, сколько при обобщенной перестановке. Метод SAG позволяет выполнить те же действия за две команды, а не за пять, которые требуются для осуществления обобщенной перестановки с помощью этого метода. Выигрыш в эффективности достигается даже в том случае, когда размер переставляемых подслов не являются степенью 2; для перестановки n подслов требуется $\lceil \log_2 n \rceil$ шагов (в n не входят возможные группы битов, расположенные на концах слова и не участвующие в перестановке).

Кроме команд SAG и *bitgather* (которые имеют имена GRP и PPERM соответственно), в [43] рассматриваются и другие возможные команды перестановки, а также перестановки путем поиска в таблицах.

7.6. Перегруппировки и преобразования индексов

Многие простые перегруппировки битов в компьютерном слове соответствуют еще более простым преобразованиям координат, или индексов битов [19]. Эти соответствия применимы к перегруппировкам элементов в любом одномерном массиве, в котором количество элементов представляет собой целую степень 2. В контексте практического программирования наиболее важен случай, когда элементы массива представляют собой слова (или превосходят их по размеру).

Рассмотрим в качестве примера внешнее идеальное перемешивание элементов массива A размером 8, дающее в результате массив B и состоящее из следующих шагов:

$A_0 \to B_0$; $A_1 \to B_2$; $A_2 \to B_4$; $A_3 \to B_6$;
$A_4 \to B_1$; $A_5 \to B_3$; $A_6 \to B_5$; $A_7 \to B_7$.

Каждый ***B*-индекс** представляет собой соответствующий А-индекс, циклически сдвинутый влево на одну позицию (при использовании циклического вращения трех битов). Обратный внешнему идеальному перемешиванию процесс получается путем циклического сдвига каждого индекса *вправо.* Ряд подобных соответствий приведен в табл. 7.1. Здесь ***n*** — количество элементов массива, а циклические сдвиги производятся над $\log_2 n$ битами.

Таблица 7.1. Перегруппировки и преобразования индексов

Перегруппировка	**Преобразование индексов**	
	Индекс массива, или прямая нумерация битов	**Обратная нумерация битов**
Реверс	Дополнение	Дополнение
Обобщенный реверс (см. раздел 7.1)	*Исключающее или* с константой	*Исключающее или* с константой
Циклический сдвиг влево на *k* позиций	Вычитание *k* (mod n)	Прибавление *k* (mod n)
Циклический сдвиг вправо на *k* **позиций**	Прибавление *k* (mod л)	Вычитание *k* (mod л)
Внешнее идеальное перемешивание	Циклический сдвиг влево на одну позицию	Циклический сдвиг вправо на одну позицию
Операция, обратная внешнему идеальному перемешиванию	Циклический сдвиг вправо на одну позицию	Циклический сдвиг влево на одну позицию
Внутреннее идеальное перемешивание	Циклический сдвиг влево на одну позицию, затем дополнение младшего бита	Дополнение младшего бита, затем циклический сдвиг вправо на одну позицию
Операция, обратная внутреннему **идеальному** перемешиванию	Дополнение младшего бита, затем циклический сдвиг вправо на одну позицию	Циклический сдвиг влево на одну позицию, затем дополнение младшего бита
Транспонирование битовой матрицы 8x8, хранящейся в 64-битовом слове	Циклический сдвиг (влево или вправо) на три позиции	Циклический сдвиг (влево или вправо) на три позиции
БПФ	Реверс битов	Реверс битов

ГЛАВА 8

УМНОЖЕНИЕ

8.1. Умножение больших чисел

Такое умножение может быть выполнено традиционным школьным способом "в столбик". Однако вместо использования массива промежуточных результатов гораздо эффективнее добавлять к произведению каждую новую строку немедленно после ее вычисления.

Если множимое состоит из m слов, множитель — из я слов, то произведение занимает не более $m+n$ слов, независимо от того, выполняется знаковое или беззнаковое умножение.

При использовании метода "в столбик" каждое 32-битовое слово рассматривается как отдельная цифра. Такой метод вполне применим, если у нас есть команда, которая дает 64-битовое произведение двух 32-битовых слов. К сожалению, даже наличие такой команды в вашем компьютере не означает доступности к ней из языка высокого уровня. В действительности многие современные RISC-компьютеры не имеют такой команды именно потому, что она недоступна из языка высокого уровня, а потому и практически не используется. (Другая причина заключается в том, что эта команда — одна из немногих, возвращающих результат в двух регистрах.)

Код данной процедуры представлен в листинге 8.1. Здесь в качестве "цифр" используются полуслова. Параметр w содержит результат, а и и v — множимое и множитель соответственно. Каждый параметр представляет собой массив полуслов, в котором первое полуслово (w[0], u[0], v[0]) является младшей значащей цифрой (прямой порядок). Параметры m и n представляют собой количество полуслов в массивах и и v соответственно.

Листинг 8.1. Знаковое умножение больших чисел

```
void mulmns(unsigned short w[], unsigned short u[],
            unsigned short v[], int m, int n) {
    unsigned int k, t, b;
    int i, j;
    for (i = 0; i < m; i++)
        W[i] = 0;
    for (j = 0; j < n; j++) {
        k = 0;
        for (i = 0; i < m; i++) {
            t = u[i]*v[j] + w[i + j] + k;
            w[i + j] = t; // (Т.е., t & 0xFFFF).
            k = t >> 16;
        }
        w[j + m] = k;
    }
    // Теперь w[] содержит беззнаковое произведение.
    // Корректируем вычитанием v*2**16m при и < 0 и
    // вычитанием u*2**16n при v < 0
    if ((short)u[m - 1] < 0) {
        b = 0; // Инициализируем заем
        for (j = 0; j < n; j++) {
            t = w[j + m] - v[j] - b;
            w[j + m] = t;
            b = t >> 31;
```

```
        }
    }
    if ((short)v[n - 1] < 0) {
        b = 0;
        for (i = 0; i < m; i++) {
            t = w[i + n] - u[i] - b;
            w[i + n] = t;
            b = t >> 31;
        }
    }
    return;
}
```

Понять код из листинга 8.1 поможет следующая иллюстрация (никакой связи между n и m нет, любое из них может быть больше другого):

	u_{m-1}	u_{m-2}	...	...	u_1	u_0
		$\times$	v_{n-1}	...	v_1	v_0
w_{m+n-1}	w_{m+n-2}	...	...	...	w_1	w_0

Данная процедура следует алгоритму М из [39, раздел 4.3.1], но в качестве рабочего языка использует С и модифицирована для выполнения знакового умножения. Заметим, что присвоение t в первой половине листинга 8.1 не может вызвать переполнения, поскольку максимальное значение, которое может быть присвоено t, равно $\left(2^{16}-1\right)^2+2\left(2^{16}-1\right)=2^{32}-1$.

Умножение больших чисел выполняется проще для беззнаковых операндов. По сути, код в листинге 8.1 выполняет беззнаковое умножение с коррекцией результата, так что, опустив код после трехстрочного комментария, можно получить процедуру беззнакового умножения больших чисел. Беззнаковую версию умножения можно распространить на знаковые операнды тремя способами.

1. Получить абсолютные значения каждого входного операнда, выполнить беззнаковое умножение, а затем изменить знак результата, если сомножители имеют разные знаки.
2. Выполнить умножение с использованием элементарных беззнаковых умножений, за исключением умножения одного из старших полуслов; в этом случае используется умножение знаковоехбеззнаковое или знаковоех знаковое.
3. Выполнить беззнаковое умножение, а затем скорректировать полученный результат.

Первый метод для вычисления абсолютного значения требует прохода по всем $m+n$ входным полусловам. Если один из операндов положителен, а второй отрицателен, то для получения дополнения отрицательного операнда и результата требуется проход по $\max(m,n)+m+n$ полусловам. Но гораздо неприятнее изменение значения входного операнда (который может быть передан в функцию по адресу), что может быть неприемлемо для ряда приложений. Как вариант обхода этой ситуации, для размещения операндов можно временно выделять дополнительную память либо после их изменения и проведения вычислений восстанавливать их предыдущие значения. Впрочем, эти обходные пути также малопривлекательны.

Второй метод требует трех вариантов элементарных операций умножения (беззнаковое х беззнаковое, беззнаковое х знаковое и знаковое х знаковое), а также знакового расширения промежуточных результатов, что существенно увеличивает время вычислений.

Таким образом, лучше выбрать третий метод. Для того чтобы понять, как именно он работает, рассмотрим умножение двух знаковых целых значений — u и v размером M и N битов соответственно. Вычисления в первой части листинга 8.1 рассматривают u как беззнаковую величину, имеющую значение $u+2^M u_{M-1}$, где u_{M-1} — знаковый бит u (т.е. $u_{M-1}=1$, если u отрицательно, и $u_{M-1}=0$ в противном случае). Аналогично, v интерпретируется программой как имеющее значение $v+2^N v_{N-1}$.

Программа вычисляет произведение этих беззнаковых величин, а именно:

$$\left(u+2^M u_{M-1}\right)\left(v+2^N v_{N-1}\right)=uv+2^M u_{M-1}v+2^N v_{N-1}u+2^{M+N} u_{M-1}v_{N-1}.$$

Для того чтобы получить требующийся результат (uv), необходимо вычесть из беззнакового произведения величину $2^M u_{M-1}v+2^N v_{N-1}u$. Вычитать член $2^{M+N}u_{M-1}v_{N-1}$ нет необходимости, поскольку известно, что результат можно выразить при помощи $M+N$ битов, так что нет необходимости вычислять биты старше, чем бит в позиции $M+N-1$. Описанные вычитания выполняются в листинге 8.1 после трехстрокового комментария и требуют прохода максимум по $m+n$ полусловам.

Может оказаться соблазнительным использовать приведенную в листинге 8.1 программу, передавая ей в качестве параметров массивы из полных слов. Такая программа будет работать на машине с прямым порядком байтов, но не с обратным. Если хранить массивы в обратном порядке, когда u [0] содержит старшее полуслово (и программа будет соответствующим образом изменена), то при передаче в качестве параметров массивов полных слов программа будет работоспособной на машине с обратным (но не с прямым) порядком байтов.

8.2. Старшее слово 64-битового умножения

Теперь рассмотрим задачу вычисления старших 32 бит произведения двух 32-битовых целых чисел. Это функции из нашего базового набора RISC-команд — *старшее слово знакового умножения* (multiply high signed, mulhs) и *старшее слово беззнакового умножения* (multiply high unsigned, mulhu).

В случае беззнакового умножения хорошо работает алгоритм из первой части листинга 8.1. Перепишем его для частного случая $m=n=2$ с развернутыми циклами и выполнением очевидных упрощений (изменив параметры на 32-битовые целые числа).

В случае знакового умножения нет необходимости в корректирующем коде из второй половины листинга 8.1. Этот код можно опустить, если правильно подойти к промежуточным результатам, объявляя их как знаковые. Конечный алгоритм приведен в листинге 8.2. При использовании беззнаковой версии просто замените все объявления int объявлениями unsigned.

Листинг 8.2. Старшее слово результата знакового умножения

```
int mulhs (int u, int v)
{
    unsigned u0, vO, w0;
    int      u1, v1, w1, w2, t;
    uO = U & 0xFFFF;   Ul = u >> 16;
    vO = v & OxFFFF;   vl = V >> 16;
    WO = u0*v0;
    t  = Ul*vO + (wO >> 16) ;
    Wl = t & OxFFFF;
```

```
    w2 = t >> 16;
    w1 = U0*v1 + w1;
    return u1*v1 + w2 + (w1 >> 16);
}
```

Данный алгоритм требует выполнения 16 базовых RISC-команд в каждой из версий (знаковой и беззнаковой), четыре из которых являются умножениями.

8.3. Преобразование знакового и беззнакового произведений

Предположим, что наш компьютер позволяет легко вычислить старшую половину 64-битового произведения двух 32-битовых *беззнаковых* целых чисел, но нам требуется выполнение этой операции для *знаковых* чисел. Можно воспользоваться процедурой из листинга 8.2, но она требует выполнения четырех умножений. Более эффективную процедуру можно найти в [6].

Анализ данного алгоритма представляет собой частный случай анализа преобразованного для умножения знаковых чисел алгоритма Кнута (см. листинг 8.1). Пусть x и y означают два 32-битовых знаковых целых числа, которые требуется перемножить. Компьютер рассматривает x как *беззнаковое* целое число, имеющее значение $x+2^{32}x_{31}$, где x_{31} — целая величина, равная 1, если x отрицательно, и 0 в противном случае. Аналогично, y рассматривается как беззнаковое число, имеющее значение $y+2^{32}y_{31}$.

Хотя нас интересуют только старшие 32 бита величины xy, компьютер производит вычисление

$$\left(x+2^{32}x_{31}\right)\left(y+2^{32}y_{31}\right)=xy+2^{32}\left(x_{31}y+y_{31}x\right)+2^{64}x_{31}y_{31}.$$

Для того чтобы получить интересующий нас результат, необходимо вычесть из полученной величины $2^{32}\left(x_{31}y+y_{31}x\right)+2^{64}x_{31}y_{31}$. Поскольку известно, что результат может быть записан при помощи 64 битов, можно использовать арифметику по модулю 2^{64}. Это означает, что можно спокойно игнорировать последний член и вычислять знаковое произведение, как показано ниже (посредством семи базовых RISC-команд).

$$
\begin{array}{ll}
p \leftarrow \mathrm{mulhu}(x,y) & \text{// Команда беззнакового 64-битового умножения} \\
t_1 \leftarrow \left(x \overset{s}{\gg} 31\right) \,\&\, y & //\ t_1 = x_{31}y \\
t_2 \leftarrow \left(y \overset{s}{\gg} 31\right) \,\&\, x & //\ t_2 = y_{31}x \\
p \leftarrow p - t_1 - t_2 & //\ p \text{ — искомый результат}
\end{array}
\qquad (1)
$$

Беззнаковое произведение

Обратное преобразование — знакового произведения в беззнаковое — выполняется так же просто. Полученная в результате программа представляет собой (1) с тем отличием, что первая команда заменяется командой знакового 64-битового умножения, а последняя операция заменяется на $p \leftarrow p+t_1+t_2$.

8.4. Умножение на константу

Совершенно очевидно, что умножение на константу можно преобразовать в последовательность *сдвигов* и *сложений*. Например, для умножения x на 13 (1101 в двоичном представлении) можно воспользоваться приведенным ниже способом (результат умножения — г).

$$t_1 \leftarrow x \ll 2$$
$$t_2 \leftarrow x \ll 3$$
$$r \leftarrow t_1 + t_2 + x$$

В этом разделе левые сдвиги записываются как умножения на степени двойки, так что приведенный план вычислений записывается как $r \leftarrow 8x + 4x + x$; это должно ясно показывать, что для вычислений на большинстве компьютеров нам потребуется четыре базовых RISC-команды.

Хотелось бы подчеркнуть, что тема умножения на константы сложнее, чем кажется на первый взгляд. Кроме прочего, при рассмотрении разных вариантов умножения следует учитывать не только количество *сдвигов* и *сложений*, необходимых для реализации того или иного умножения. Для иллюстрации сказанного рассмотрим два варианта умножения на 45 (101101 в двоичном представлении).

$t \leftarrow 4*$	$t_1 \leftarrow 4*$
$r \leftarrow x + t$	$t_2 \leftarrow 8x$
$t \leftarrow 2t$	$t_3 \leftarrow 32x$
$r \leftarrow r + t$	$r \leftarrow t_1 + x$
$t \leftarrow 4t$	$t_3 \leftarrow t_3 + t_2$
$r \leftarrow r + t$	$r \leftarrow r + t_3$

План вычислений, представленный слева, использует переменную t *для* хранения величины x, сдвинутой на количество позиций, соответствующих единичным битам множителя. Каждое сдвинутое значение получается из сдвинутого перед этим. Такой план обладает следующими преимуществами:

- для работы кроме регистров для входного и выходного значения требуется только один регистр;
- все команды, за исключением первых двух, двухадресные;
- величины сдвигов относительно невелики.

Эти же свойства остаются действительными и для других множителей.

В схеме справа сначала выполняются все *сдвиги* с x в качестве операнда. Главное преимущество такого решения — увеличение параллелизма вычислений. На компьютере с достаточными возможностями распараллеливания вычислений на уровне команд правая схема выполняется всего за три такта, в то время как левая схема на машине с неограниченными возможностями распараллеливания вычислений на уровне команд требует четыре такта.

Кроме того, задача поиска минимального количества команд (подразумевая под ними команды *сдвига* и *сложения)* для выполнения умножения на константу нетривиальна. Далее будем считать, что множество допустимых команд состоит из *сложения, вычитания, сдвига влево* и *отрицания (не).* Кроме того, будем считать, что формат команд — трехадресный. Впрочем, если ограничиться только *сложением* (при этом сдвиг влево осуществляется путем сложения числа с самим собой, затем суммы с самой собой и т.д.), задача отнюдь не упрощается, как и в случае, если в набор команд добавить другие ко-

манды, например команду HP PA-RISC *сдвиг и сложение* (она сдвигает содержимое регистра на одну, две или три позиции, складывает с содержимым второго регистра и помещает результат в третий регистр; таким образом можно осуществить умножение на 3, 5 или 9 одной, по-видимому, очень быстрой командой). Будем также считать, что нас интересуют только младшие 32 бита произведения.

Первое улучшение к предложенной выше базовой схеме бинарного разложения состоит в использовании *вычитания* для сокращения последовательности вычислений, если множитель содержит группу из трех или большего количества последовательных единичных битов. Например, для умножения на 28 (11100 в двоичном представлении) можно вычислить $32x - 4x$ (три команды) вместо того, чтобы вычислять $16x + 8x + 4x$ за пять команд. На машинах с использованием дополнения до двойки конечный результат остается корректным, даже если промежуточное значение $32x$ вызывает переполнение.

Умножение на константу *m* по схеме бинарного разложения с использованием только команд *сдвига* и *сложения* требует выполнения $2\text{pop}(m) - 1 - \delta$ команд, где $\delta = 1$, если *m* завершается единичным битом (нечетно), и равно 0 в противном случае. Если же в дополнение к перечисленным командам используются команды вычитания, то умножение требует выполнения $4g(m) + 2s(m) - 1 - 3$ команд, где $g(m)$ — количество групп из двух и более последовательных единичных битов в *m*, а $s(m)$ — количество "одиноких" единичных битов. Значение δ определяется так же, как и ранее.

Для группы из двух байтов оба метода равноценны.

Следующее улучшение состоит в рассмотрении групп специализированного вида, состоящих из разделенных одним нулевым битом групп из последовательных единичных битов. Рассмотрим, например, от=55 (110111 в двоичном представлении). Метод групп позволяет вычислить произведение путем выполнения шести команд: $(64x - 16x) + (8x - x)$. Если же вычислять произведение как $64x - 8x - x$, то потребуется выполнение только четырех команд. Аналогично, умножение на двоичное число 1 1011 1011 требует выполнения шести команд, что иллюстрируется формулой $512x - 64x - 4x - x$.

Приведенные выше формулы дают верхнюю границу количества команд, требующихся для умножения переменной *x* на данное число от. Другая граница может быть получена исходя из размера *m* в битах: $n = \lfloor \log_2 m \rfloor + 1$.

Теорема. *Умножение переменной x на n-битовую константу m, $m \geq 1$, может быть выполнено не более чем за n команд сложения, вычитания и сдвига влево на некоторое заданное значение.*

Доказательство (по индукции по и). Умножение на 1 выполняется за 0 команд, так что для $n = 1$ теорема справедлива. Для $n > 1$ умножение на от (если *m* заканчивается нулевым битом) можно реализовать путем умножения на число, представляющее собой левые $n - 1$ битов числа m (т.е. на $m/2$) за $n - 1$ команду, и *сдвига влево* на одну позицию, т.е. всего за *n* команд.

Если *m* в двоичном исчислении заканчивается на 01, то mx можно вычислить путем умножения *x* на число, состоящее из левых $n - 2$ битов от (для чего потребуется не более $n - 2$ команд), с последующим *сдвигом* результата *влево* на две позиции и *сложения* с *x*. Всего для этого требуется выполнение *n* команд.

Если от в двоичном исчислении заканчивается на И, то нужно рассмотреть случаи, когда от в двоичном представлении заканчивается на ООП, 0111, 1011 и 1111. Пусть

t — результат умножения x на левые $n-4$ бита m. Если m заканчивается на 0011, то $mx = 16t + 2x + x$, для чего требуется выполнение $(n-4)+4 =$ я команд. Если m заканчивается на 0111, то $mx = 16t + 8x - x$, а если на 1111 — то $mx = 16t + 16x - x$; в обоих случаях для вычислений требуется выполнить n команд. Последний случай — когда двоичное представление m заканчивается на 1011.

При этом легко показать, что mx можно вычислить за n команд, если m заканчивается на 001011, 011011 или 111011. Остается рассмотреть случай 101011.

Эти рассуждения можно продолжать — и всякий раз "последний случай" будет представлять собой число вида 101010... 10101011. В конце концов будет достигнут размер m и нам останется рассмотреть только n-битовое число 101010... 10101011, которое содержит $n/2+1$ единичных битов. Но ранее было показано, что x на такое число можно умножить за $2(n/2+1)$-$2 = n$ команд.

Таким образом, на 32-битовом компьютере, например, умножение на любую константу может быть выполнено максимум за 32 команды в соответствии с описанным выше методом.

Проверка показывает, что, когда n четно, n-битовое число 101010... 101011 требует выполнения и команд, как и число 1010101...0101 10 в случае нечетного n, так что указанная граница компактна.

Описанная методология не слишком сложна до тех пор, пока мы работаем с ней вручную или встраиваем в алгоритм для использования, например, в компиляторе. Однако такой алгоритм не всегда дает оптимальный код, поскольку иногда возможно дальнейшее его улучшение. Это улучшение может использовать, например, результаты разложения m или промежуточные результаты вычислений. Взглянем еще раз на случай m=45. Описанный выше метод требует выполнения шести команд, но разложение 45 на множители 5 и 9 дает нам решение с использованием всего четырех команд.

$$t \leftarrow 4x + x$$
$$r \leftarrow 8t + t$$

Разложение может быть скомбинировано с аддитивными методами. Например, умножение на 106 при использовании аддитивного метода требует выполнения семи команд, но запись 106 как $7 \cdot 15 + 1$ дает решение, состоящее из пяти команд.

Решение задачи о максимальном количестве команд, необходимом для умножения на n-битовую константу с использованием разложения, автору не известно. Для больших n оно может быть меньше, чем доказанная ранее граница n. Например, значение m=0xAAAAAAAB без использования разложения требует 32 команд, но если записать это значение как $2 \cdot 5 \cdot 17 \cdot 257 \cdot 65537 + 1$, то будет получено решение, состоящее из 10 команд (что, вероятно, нетипично для больших чисел; разложение отражает чередование единичных и нулевых битов в двоичном представлении числа).

В [39, раздел 4.6.3] рассматривается тесно связанная с данной задачей проблема вычисления a^m с использованием минимального количества умножений. Эта задача аналогична задаче умножения на m с использованием только команд сложения. Алгоритм вычисления тес , который можно использовать в компиляторе, описан в [5].

ГЛАВА 9
ЦЕЛОЧИСЛЕННОЕ ДЕЛЕНИЕ

9.1. Предварительные сведения

В этой и следующей главах приводится ряд приемов и алгоритмов "компьютерного деления" целых чисел. В математических формулах используется запись x/y для обозначения обычного деления, $x \div y$ — для знакового компьютерного деления целых чисел (с отсечением дробной части), и $x \overset{u}{\div} y$ — для беззнакового компьютерного деления целых чисел. В коде на языке С x/y означает компьютерное деление, которое может быть как знаковым, так и беззнаковым — в зависимости от его операндов.

Деление представляет собой сложный процесс, и включающие его алгоритмы зачастую не слишком элегантны. Более того, серьезным вопросом является даже то, как именно должно быть определено знаковое деление. Большинство языков высокого уровня и наборов команд компьютера определяют результат целочисленного деления как результат рационального деления с отброшенной дробной частью. Ниже проиллюстрированы данное и два других возможных определения целочисленного деления (rem означает остаток от деления).

		Отсечение	Модуль	Округление к меньшему значению
7÷3	=	2 rem 1	2 rem 1	2 rem 1
(-7)÷3	=	-2 rem -1	-3 rem 2	-3 rem 2
7÷(-3)	=	-2 rem 1	-2 rem 1	-3 rem -2
(-7)÷(-3)	=	2 rem -1	3 rem 2	2 rem -1

Соотношение *Делимое* = *Частное* * *Делитель* + *Остаток* выполняется при любом определении целочисленного деления. "Модульное" деление определяется как требующее, чтобы остаток от деления был неотрицательным[1], а деление с округлением к меньшему значению требует, чтобы результат представлял собой наибольшее целое число, не превосходящее результата рационального деления. При положительном делителе эти два метода дают одинаковые результаты. Есть и четвертый, редко используемый вариант целочисленного деления, когда результат рационального деления округляется до ближайшего целого числа.

Одно из достоинств модульного деления и деления с округлением к меньшему значению заключается в том, что они упрощают ряд используемых приемов. Например, деление на 2^n можно заменить *знаковым сдвигом вправо* на и позиций, а остаток от деления x на 2^n получается в результате применения операции *и к х к* 2" −1. Я также подозреваю, что модульное деление и деление с округлением к меньшему значению чаще дают тот результат, который вас интересует. Например, предположим, что вы пишете программу

---

[1] Я знаю, что меня будут критиковать за такую классификацию, поскольку из понятия "модуль" не следует понятие "неотрицательный". Оператор mod у Кнута [38] означает остаток от деления с округлением к меньшему значению (floor), который имеет отрицательное значение (или равен 0), если делитель отрицателен. Ряд языков программирования определяет функцию mod как остаток при отсекающем делении. Однако, например, в теории комплексных чисел модуль есть величина неотрицательная, как и в ряде других разделов математики.

для вывода графика целочисленной функции, значения которой находятся в диапазоне от *imin* до *imax*, и хотите, чтобы крайними значениями по оси ординат были наименьшие кратные 10 числа, включающие *imin* и *imax*. Эти крайние значения равны $(imin \div 10) * 10$ и $((imax + 9) \div 10) * 10$, если воспользоваться "модульным" делением или делением с округлением к меньшему значению. Если же использовать обычное деление, вам придется писать код наподобие следующего:

```
if  (imin >= 0)  gmin =  (imin/10)*10;
else             gmin =  ((imin -  9)/10)*10;
if  (imax >= 0)  gmax =  ((imax +  9)/10)*10;
else             gmax =  (imax/10)*10;
```

Помимо того что "модульное" деление и деление с округлением к меньшему значению практичнее деления с отсечением, следует отметить, что неотрицательный остаток от деления требуется, пожалуй, чаще, чем остаток, который может принимать отрицательные значения.

Сделать выбор между "модульным" делением и делением с округлением к меньшему значению непросто, поскольку они отличаются только в случае отрицательного делителя, что встречается довольно редко. Обращение к существующим языкам программирования высокого уровня в этой ситуации не поможет, поскольку практически везде деление знаковых целых чисел x/y означает деление с отсечением. Некоторые языки программирования при делении целых чисел возвращают число с плавающей точкой, т.е. рациональное. Ничуть не проще ситуация с остатками от деления. В Fortran 90 функция MOD дает остаток от деления с отсечением, а функция MODULO — остаток от деления с округлением к меньшему значению (который может быть отрицательным). Аналогично в Common Lisp и ADA функция REM дает остаток от деления с отсечением, а MOD — остаток от деления с округлением к меньшему значению. В PL/I значение MOD неотрицательно (остаток от модульного деления). В Pascal операция A mod B определена только для B > о, а ее результат всегда неотрицателен (т.е. это остаток от модульного деления либо от деления с округлением к меньшему значению).

Как бы то ни было, мы не можем изменить мир, даже если бы и хотели это сделать[2], так что далее, следуя за большинством, будем использовать обычное (отсекающее) определение деления $x \div y$.

Самое ценное свойство отсекающего деления заключается в том, что для $d \neq 0$ выполняются следующие соотношения:

$$(-n) \div d = n \div (-d) = -(n \div d) .$$

Однако к подобным преобразованиям в программах следует подходить осторожно, так как если n или d представляют собой наибольшее (по модулю) отрицательное число, то $-n$ или $-d$ не могут быть представлены 32 битами. Операция $(-2^{32}) \div (-1)$ вызывает переполнение (результат не может быть представлен в виде знаковой величины в дополнительном коде), и на большинстве компьютеров результат окажется неопределенным либо операция — невыполненной.

---

[2] Впрочем, некоторые пытаются. Язык IBM PL.8 использует модульное деление, а команда деления в машине Кнута MMIX использует деление с округлением к меньшему значению [48].

Знаковое целое (отсекающее) деление связано с обычным рациональным делением следующим соотношением:

$$n \div d = \begin{cases} \left\lfloor \frac{n}{d} \right\rfloor, & \text{если } d \neq 0,\ nd \geq 0; \\ \left\lceil \frac{n}{d} \right\rceil, & \text{если } d \neq 0,\ nd < 0. \end{cases} \qquad (1)$$

Беззнаковое целое деление, т.е. деление, при котором и n и d рассматриваются как беззнаковые целые числа, удовлетворяет верхней части (1).

В дальнейшем рассмотрении темы используется ряд элементарных арифметических свойств, которые приводятся здесь без доказательства. Функции, возвращающие наибольшее целое число, меньшее данного (floor — "пол"), и наименьшее, большее данного (ceil — "потолок"), подробно описаны в [18, 38].

Теорема D1. *Для действительного x и целого k справедливы следующие соотношения:*

$$\lfloor x \rfloor = -\lceil -x \rceil \qquad \lceil x \rceil = -\lfloor -x \rfloor$$
$$x-1 < \lfloor x \rfloor \leq x \qquad x \leq \lceil x \rceil < x+1$$
$$\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1 \qquad \lceil x \rceil - 1 < x \leq \lceil x \rceil$$
$$x \geq k \Leftrightarrow \lfloor x \rfloor \geq k \qquad x \leq k \Leftrightarrow \lceil x \rceil \leq k$$
$$x > k \Rightarrow \lfloor x \rfloor > k \qquad x < k \Rightarrow \lceil x \rceil < k$$
$$x \leq k \Rightarrow \lfloor x \rfloor \leq k \Rightarrow x < k+1 \qquad x \geq k \Rightarrow \lceil x \rceil \geq k \Rightarrow x > k-1$$
$$x < k \Leftrightarrow \lfloor x \rfloor < k \qquad x > k \Leftrightarrow \lceil x \rceil > k$$

Теорема D2. *Для целых n и d $(d > 0)$ справедливы соотношения:*

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d+1}{d} \right\rceil \quad и \quad \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d-1}{d} \right\rfloor.$$

При $d<0$ справедливы соотношения

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d-1}{d} \right\rceil \quad и \quad \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d+1}{d} \right\rfloor.$$

Теорема D3. *Для действительного x и целого $d \neq 0$ справедливы соотношения*

$$\lfloor \lfloor x \rfloor / d \rfloor = \lfloor x/d \rfloor \quad и \quad \lceil \lceil x \rceil / d \rceil = \lceil x/d \rceil.$$

Следствие. *Для действительных чисел a и b $(b \neq 0)$ и целого $d \neq 0$ справедливы соотношения*

$$\left\lfloor \left\lfloor \frac{a}{b} \right\rfloor \Big/ d \right\rfloor = \left\lfloor \frac{a}{bd} \right\rfloor \quad и \quad \left\lceil \left\lceil \frac{a}{b} \right\rceil \Big/ d \right\rceil = \left\lceil \frac{a}{bd} \right\rceil.$$

Теорема D4. *Для целых n и d $(d \neq 0)$ и действительного x справедливы соотношения*

$$\left\lfloor \frac{n}{d} + x \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor, \ \text{если } 0 \leq x < \left| \frac{1}{d} \right|, \quad и \quad \left\lceil \frac{n}{d} + x \right\rceil = \left\lceil \frac{n}{d} \right\rceil, \ \text{если } -\left| \frac{1}{d} \right| < x \leq 0.$$

В приведенной ниже теореме $\text{rem}(n,d)$ обозначает остаток от деления n на d. Для отрицательных значений d действует следующее определение: $\text{rem}(n,-d)=\text{rem}(n,d)$, как в случае деления с отсечением и модульного деления. При $n<0$ функция $\text{rem}(n,d)$ не используется (таким образом, при нашем использовании данной функции ее значения всегда неотрицательны).

Теорема D5. *Для $n \geq 0$ и $d \neq 0$ справедливы соотношения:*

$$\text{rem}(2n,d)=\begin{cases}2\,\text{rem}(n,d) & \text{или}\\ 2\,\text{rem}(n,d)-|d|\end{cases} \qquad \text{и} \quad \text{rem}(2n+1,d)=\begin{cases}2\,\text{rem}(n,d)+1 & \text{или}\\ 2\,\text{rem}(n,d)-|d|+1\end{cases}$$

(причем каждое из значений больше или равно 0 и меньше $|d|$).

Теорема D6. *Для $n \geq 0$ и $d \neq 0$ справедливо соотношение:*

$$\text{rem}(2n,2d)=2\,\text{rem}(n,d).$$

Теоремы D5 и D6 легко доказываются исходя из базового определения остатка, т.е. что для некоторого целого q он удовлетворяет соотношению

$$n=qd+\text{rem}(n,d),\quad 0 \leq \text{rem}(n,d)<|d|,$$

где $n \geq 0$ и $d \neq 0$ (и и d могут быть не целыми, однако будем применять наши теоремы только к целым числам).

9.2. Деление больших чисел

Как и в случае умножения, деление больших чисел можно осуществить традиционным школьным способом "в столбик". Однако детали реализации такого метода на удивление сложны. В листинге 9.1 приведен алгоритм Кнута [39, раздел 4.3.1], реализованный на языке С. В его основе лежит беззнаковое деление типа $32 \div 16 \Rightarrow 32$ (на самом деле частное при такой операции деления имеет длину не более 17 бит).

Листинг 9.1. Беззнаковое деление больших целых чисел

```
int divmnu (unsigned short q[], unsigned short r[],
            const unsigned short u[],
            const unsigned short v[],
            int m, int n)
{
    const unsigned b = 65536; // Основание чисел (16 битов)
    unsigned short *un, *vn;  // Нормализованный вид и и v
    unsigned qhat;       // Предполагаемая цифра частного
    unsigned rhat;       // Остаток
    unsigned p;          // Произведение двух цифр
    int s, i, j, t, k;
    if (m < n || n <= 0 || v[n-1] == 0)
        return 1; // Возвращается при некорректном параметре
    if (n == 1)   // Частный случай делителя из одной цифры
    {
        k = 0;
        for (j = m - 1; j >= 0; j--)
        {
```

```
            q[j] = (k*b + u[j])/v[0];
            k = (k*b + u[j]) - q[j]*v[0] ;
        }
        if (r != NULL) r[0] = k;
        return 0;
    }
    // Нормализация путем сдвига v влево, такого, что
    // старший бит становится единичным, и сдвига u влево
    // на ту же величину. Нам может потребоваться добавить
    // старшую цифру к частному; мы делаем это безусловно
    s = nlz(v[n-1]) - 16; // 0 <= s <= 16.
    vn = (unsigned short *)alloca(2*n);
    for (i = n - 1; i > 0; i--)
        vn[i] = (v[i] << s) | (v[i-l] >> 16-s);
    vn[0] = v[0] << s;
    un = (unsigned short *)alloca(2*(m + 1));
    un[m] = u[m-l] >> 16-s;
    for (i = m - 1; i > 0; i--)
        un[i] = (u[i] << s) | (u[i-1] >> 16-s);
    un[0] = u[0] << S;
    for (j = m - n; j >= 0; j--)    // Главный цикл
    {                               // Вычисляем оценку q[j]
        qhat = (un[j+n]*b + un[j+n-1])/vn[n-1];
        rhat = (un[j+n]*b + un[j+n-1]) - qhat*vn[n-1];
    again:
        if (qhat >= b || qhat*vn[n-2] > b*rhat + untj+n-2])
        {
            qhat = qhat - 1;
            rhat = rhat + vn[n-1];
            if (rhat < b) goto again;
        }
        // Умножение и вычитание
        k = 0;
        for (i = 0; i < n; i++)
        {
            p = qhat*vn[i];
            t = un[i+j] - k - (p & 0xFFFF);
            un[i+j] = t;
            k = (p >> 16) - (t >> 16);
        }
        t = un[j+n] - k;
        un[j+n] = t;
        q[j] = qhat;   // Сохранение цифры частного
        if (t < 0)     // Если мы вычли слишком много,
        {              // вернем назад
            qtj] = q[j] - 1;
            k = 0;
            for (i = 0; i < n; i++)
            {
                t = un[i+j] + vn[i] + k;
                un[i+j] = t;
                k = t >> 16;
            }
            un[j+n] = untj+n] + k;
        }
    } // for j
    // Если вызывающей функции нужно значение остатка,
    // денормализуем и возвращаем его
    if (r != NULL)
```

```
    {
        for (i = 0; i < n; i++)
            r[i] = (un[i] >> s) | (un[i+1] << 16-s);
    }
    return 0;
}
```

Алгоритм обрабатывает входные и выходные данные по полслова. Конечно, было бы предпочтительнее обрабатывать данные по слову, но такой алгоритм требует наличия команды деления $64 \overset{u}{\div} 32 \Rightarrow 32$. Однако предполагается, что такой команды в компьютере нет (или она недоступна из языка высокого уровня). Хотя при этом предполагается наличие команды деления $32 \overset{u}{\div} 32 \Rightarrow 32$, для решения нашей задачи достаточно команды $32 \overset{u}{\div} 16 \Rightarrow 16$.

Таким образом, в данной реализации алгоритма Кнута основание счисления b равно 65536. В [39] вы найдете подробное пояснение данного алгоритма.

Делимое и и делитель v используют прямой порядок, т.е. и [0] и v [0] представляют собой младшие цифры числа (впрочем, данный код корректно работает при использовании как прямого, так обратного порядка). Параметры m и n представляют собой количество полуслов в и и v соответственно (Кнут определяет m как длину частного). Вызывающая функция должна обеспечить память для хранения частного q и (необязательно) для остатка r. Пространство для частного должно иметь размер, как минимум, m-n+1 полуслов и n полуслов для остатка. Если значение остатка нас не интересует, вместо адреса для его размещения можно передать параметр NULL.

Алгоритм требует, чтобы старшая цифра делителя v[n-1] была ненулевой. Это упрощает нормализацию и помогает убедиться, что вызывающая функция выделила достаточное количество памяти для частного. Код проверяет, является ли значение v[n-1] ненулевым, а также выполнение условий n > 1 и m > п. Если любое из этих условий нарушено, возвращается код ошибки (значение 1).

После этих проверок код выполняет деление для частного случая, когда делитель имеет длину 1. Это выделение частного случая не для ускорения работы; просто оставшаяся часть алгоритма требует, чтобы длина делителя была не менее 2.

Если делитель имеет длину 2 или большую, алгоритм нормализует делитель, сдвигая его влево на величину, достаточную, чтобы старший бит делителя был равен 1. На ту же величину сдвигается и делимое, так что эти сдвиги никак не влияют на величину частного. Как поясняется у Кнута, эти шаги необходимы для того, чтобы облегчить оценку каждой цифры частного с хорошей точностью. Для определения величины сдвига используется функция *количества ведущих нулевых битов* $\text{nlz}(x)$.

При выполнении нормализации для делимого и делителя выделяется новая память. Это делается в связи с тем, что с точки зрения вызывающей функции крайне нежелательно изменение входных данных, которые вообще могут оказаться константами, расположенными в памяти, доступной только для чтения. Кроме того, из-за сдвига делимому может понадобиться дополнительное полуслово для хранения старшей цифры. Для выделения этой памяти в С идеально подходит функция alloca(), которая обычно выделяет память с помощью двух-трех команд и не требует явного ее освобождения. Эта память выделяется в программном стеке, так что она автоматически освобождается при возврате из функции.

В основном цикле происходит быстрое вычисление цифр частного, по одной цифре за итерацию; делимое при этом уменьшается и в конечном счете становится равным остатку. Оценка значения очередной цифры частного qhat после уточнения в цикле, помеченном меткой again, всегда оказывается либо точной, либо больше точного значения на 1.

Следующие шаги состоят в умножении qhat на делитель и вычитании произведения из полученного остатка, как и в методе деления столбиком. Если остаток оказывается отрицательным, необходимо уменьшить цифру частного на 1 и либо заново вычислить произведение и вычесть его из остатка, либо, что гораздо проще, прибавить к отрицательному значению остатка делитель. Такой шаг делается не более одного раза, поскольку цифра частного либо точна, либо больше точного значения на 1.

Последнее действие состоит в возврате остатка вызывающей функции, если переданный адрес памяти для хранения остатка не NULL. Остаток при этом должен быть сдвинут вправо на ту же величину, на которую в целях нормализации сдвигались влево делитель и делимое.

Шаги, связанные с неверной оценкой цифры частного, выполняются достаточно редко. Чтобы убедиться в этом, заметим, что первое вычисление оценки каждой цифры частного `qhat` выполняется путем деления двух старших цифр текущего остатка на старшую цифру делителя. Шаги цикла again уточняют оценку путем деления трех старших цифр текущего остатка на *две* старшие цифры делителя (доказательство опущено; убедиться в корректности этих действий можно, проведя несколько тестовых вычислений с основанием системы счисления $b = 10$). Заметим, что из-за выполненной нормализации делитель имеет значение, не меньшее $b/2$, а делимое не более чем в b раз превышает делитель (поскольку любой остаток меньше делителя).

Насколько точна оценка частного при использовании только трех цифр делимого и двух — делителя? Можно показать, что в связи с выполненной нормализацией такая оценка довольно точна. Чтобы увидеть это в какой-то мере интуитивно (без формального доказательства), рассмотрим оценку u/v в арифметике с основанием счисления 10. Можно показать, что такая оценка всегда несколько завышена (или точна). Следовательно, наихудшим случаем является тот, при котором усечение делителя до двух цифр максимально уменьшает его, а усечение делимого до трех цифр не уменьшает его величину, которая должна быть максимально возможной. Это происходит в случае 49900...0/5099...9, что приводит к оценке 499/50=9.98. Точный результат составляет примерно 499/51=9.7843. Разница в 0.1957 показывает, что оценка цифры частного отличается от истинного значения не более чем на 1, причем это происходит примерно в 20% случаев (в предположении, что цифры частного распределены равномерно). Это, в свою очередь, означает, что дополнительные действия, связанные с неточной оценкой цифры частного, будут выполняться примерно в 20% случаев.

Проведение этого (нестрогого) анализа для общего случая системы счисления b приводит к выводу о том, что оценочное и истинное значение цифры частного отличается не более чем на $2/b$. В случае $b=65536$ мы получим, что оценочное и истинное значение цифры частного отличаются не более чем на 1 и происходит это с вероятностью примерно 2/65536=0.00003. Следовательно, дополнительные действия выполняются только примерно для 0.003% цифр частного.

В качестве конкретного примера, когда требуются дополнительные корректирующие действия, в десятичной системе счисления можно привести деление 4500/501. Аналогичный пример для системы счисления по основанию 65536 — 0x7FFF800000000000/0x800000000001.

Не будем пытаться оценить время работы программ в целом, ограничимся лишь замечанием, что для больших m и n время выполнения определяется циклом умножения/вычитания. Хороший компилятор в состоянии использовать для этого цикла всего 16 базовых RISC-команд, одна из которых — *умножение*. Цикл `for` j выполняется n раз, а цикл умножения/вычитания — $m-n+1$ раз, что дает время выполнения этой части программы, равное $(15+mul)n(m-n+1)$ тактов, где mul — время умножения двух 16-

битовых переменных. Кроме того, в программе выполняется $m-n+1$ команд деления и одна — вычисления *количества ведущих нулевых битов*.

Знаковое деление больших чисел

Приводить специализированный алгоритм для знакового деления больших чисел нет необходимости, просто укажем, как можно адаптировать для этого алгоритм беззнакового деления больших чисел.

1. Изменить знак у делимого, если оно отрицательно. Сделать то же для делителя.
2. Преобразовать делимое и делитель в беззнаковое представление.
3. Использовать алгоритм беззнакового деления больших чисел.
4. Преобразовать частное и остаток в знаковое представление.
5. Изменить знак частного, если знаки делимого и делителя были различны.
6. Если делимое отрицательно, изменить знак остатка.

Иногда выполнение этих шагов требует добавления или удаления старшего бита. Например, предположим для простоты, что числа представлены в системе счисления с основанием 256 (один байт на цифру) и что при знаковом представлении числа старший бит последовательности цифр является знаковым. Такое представление наиболее похоже на обычное представление чисел в дополнительном коде. В этом случае делитель 255, который в знаковом представлении имеет вид 0x00FF, при выполнении шага 2 должен быть сокращен до 0xFF. Аналогично, если частное из шага 3 начинается с единичного бита, для корректного представления в виде знакового числа нужно добавить к нему ведущий нулевой байт.

9.3. Беззнаковое короткое деление на основе знакового

Под "коротким делением" подразумевается деление одного слова на другое (т.е. деление типа $32 \div 32 \Rightarrow 32$). Это обычное деление, которое в языке С и многих других языках программирования высокого уровня выполняется оператором "/" с целыми операндами. В С имеется и знаковое и беззнаковое короткое деление, но в ряде компьютеров в набор команд входит только знаковое деление. Каким образом можно реализовать беззнаковое деление на такой машине? Непохоже, чтобы такая задача решалась гладко и просто, тем не менее рассмотрим некоторые возможные варианты.

Использование знакового длинного деления

Даже если компьютер оснащен знаковым длинным делением ($64 \div 32 \Rightarrow 32$), осуществление беззнакового короткого деления не такая простая задача, как может показаться на первый взгляд. В компиляторе XLC для IBM RS/6000 $q \leftarrow \left\lfloor n \overset{u}{\div} d \right\rfloor$ реализуется следующим образом (запись с использованием псевдокода).

1. if $n \overset{u}{<} d$ then $q \leftarrow 0$
2. else if $d = 1$ then $q \leftarrow n$
3. else **if rf<l** then $q \leftarrow 1$
4. else $q \leftarrow (0 \| n) \div d$

Третья строка в действительности проверяет выполнение условия $d \overset{u}{\geq} 2^{31}$. Если d в этом месте алгебраически меньше или равно 1, то, поскольку оно не равно 1 (эта проверка выполнялась во второй строке), алгебраически оно должно не превышать 0. Нас не беспокоит случай $d=0$, поэтому, если проверяемое в третьей строке условие истинно, это означает, что установлен знаковый бит d, т.е. $d \overset{u}{\geq} 2^{31}$. Поскольку из первой строки известно, что $n \overset{u}{\geq} d$, и и не может превышать $2^{32}-1$, то $n \overset{u}{\div} d = 1$.

Запись в четвертой строке означает формирование целого числа двойной длины, состоящего из 32 нулевых битов, за которыми следует 32-битовая величина n, и деление его на d. Проверка $d=1$ во второй строке необходима для того, чтобы гарантировать, что это деление не вызовет переполнения (переполнение может возникнуть при $n \overset{u}{\geq} 2^{31}$, и частное в этом случае оказывается неопределенным).

Если объединить сравнения во второй и третьей строках[3], то описанный выше алгоритм может быть реализован при помощи 11 команд, три из которых — команды ветвления. Если необходимо выполнение деления при $d=0$ (завершающееся генерацией прерывания), то третью строку можно заменить строкой "`else if` $d<0$ `then` $q \leftarrow 1$", что приводит к 12-командной реализации на машине RS/6000.

Не так сложно изменить код таким образом, чтобы наиболее вероятные значения ($2 \overset{u}{\leq} d \overset{u}{<} 2^{31}$) не требовали такого большого количества проверок (начав код с проверки `if` $d<1$), но это приведет к некоторому увеличению объема получаемого кода.

Использование знакового короткого деления

Если знаковое длинное деление недоступно, но есть знаковое короткое деление, то $n \overset{u}{\div} d$ можно реализовать приведением задачи к случаю $n,d < 2^{31}$ и использованием машинной команды *деления*. Если $d \overset{u}{\geq} 2^{31}$, то $n \overset{u}{\div} d$ может быть только 0 или 1, так что от этого условия легко освободиться. Далее можно воспользоваться тем фактом, что выражение $\left(\left(n \overset{u}{\div} 2\right) \div d\right) \times 2$ приближенно равно значению $n \overset{u}{\div} d$ с ошибкой, равной 0 или 1. Это приводит нас к следующему методу (записан с использованием псевдокода).

1. if $d<0$ then if $n \overset{u}{<} d$ then $q \leftarrow 0$
2. else $q \leftarrow 1$
3. else do
4. $q \leftarrow \left(\left(n \overset{u}{\div} 2\right) \div d\right) \times 2$
5. $r \leftarrow n - qd$
6. if $r \overset{u}{\geq} d$ then $q \leftarrow q+1$
7. end

[3] Выполнение команды *сравнения* на RS/6000 устанавливает биты состояния, указывающие выполнение отношений меньше, больше и равно.

Проверка $d<0$ *в* первой строке на самом деле выясняет, не выполняется ли условие $d \overset{u}{\geq} 2^{31}$. Если это условие истинно, то наибольшее частное может быть равно только $(2^{31}-1)\div 2^{31}=1$, так что первые две строки представляют собой вычисление частного для случая $d \overset{u}{>} 1$.

Четвертая строка представляет собой код, выполняющий команды *беззнакового сдвига вправо на 1 бит, деления, сдвига влево на 1 бит.* Понятно, что $n \overset{u}{\div} 2 \overset{u}{<} 2^{31}$, и к этому моменту мы убедились, что и $d \overset{u}{<} 2^{31}$, а потому эти величины могут использоваться в машинной команде знакового деления (если $d=0$, машина сообщит о переполнении). Вычисления в четвертой строке дают следующую оценку частного (с использованием следствия из теоремы D3):

$$q=\lfloor\lfloor n/2\rfloor/d\rfloor\cdot 2=\lfloor n/(2d)\rfloor=\frac{n-\text{rem}(n,2d)}{d}.$$

В строке 5 вычисляется соответствующий этому частному остаток:

$$r=n-\frac{n-\text{rem}(n,2d)}{d}d=\text{rem}(n,2d).$$

Таким образом, $0\leq r<2d$. Если $r<d$, то q представляет собой корректный остаток. Если же $r\geq d$, то верное значение частного получается добавлением 1 к вычисленному в строке 4 значению (при выполнении проверки значения остатка программа должна использовать беззнаковое сравнение, так как возможна ситуация, когда $r > 2^n$).

Загружая значение 0 в q до выполнения сравнения $n \overset{u}{<} d$ и кодируя присвоение $q\leftarrow 1$ в строке 2 как переход к присвоению $q\leftarrow q+1$ в строке 6, получим 14 команд на большинстве машин, четыре из которых будут представлять собой команды ветвления. Достаточно просто дополнить данный код для вычисления не только частного, но и остатка от деления. Для этого необходимо добавить к строке 1 присвоение $r\leftarrow n$, к строке 2 — $r\leftarrow n-d$, а к части "then" строки 6 — $r\leftarrow r-d$ (либо, если пойти на использование команды умножения, можно обойтись единственным добавлением присвоения $r\leftarrow n-qd$ после выполнения всех вычислений).

Альтернативой строкам 1 и 2 являются строки

$$\begin{array}{l} \textit{if}\ \ n \overset{u}{<} d\ \ \text{then}\ \ q\leftarrow 0 \\ \text{else}\ \ \text{if}\ \ \textbf{rf}<0\ \ \text{then}\ \ q\leftarrow 1, \end{array}$$

что приводит к более компактному коду, состоящему из 13 команд, три из которых являются командами ветвления. Однако в этом случае для наиболее вероятной ситуации (небольшие величины, $n>d$) будет выполняться большее количество команд.

Используя выражения предикатов, программу можно переписать следующим образом.

1. **if rf<0** then $q\leftarrow\left(n \overset{u}{\geq} d\right)$
2. else do
3. $\quad q\leftarrow\left(\left(n \overset{u}{\div} 2\right)\div d\right)\times 2$
4. $\quad r\leftarrow n-qd$

5. $q \leftarrow q + \left(r \overset{u}{\geq} d \right)$

6. **end**

Это экономит две команды ветвления (если в компьютере имеется возможность вычисления указанных предикатов без использования ветвления). На Compaq Alpha предикаты из этого алгоритма вычисляются одной командой (CMPULE), на MIPS требуется выполнение двух команд (SLTU, XORI). На большинстве компьютеров вычисление каждого из этих предикатов требует выполнения четырех команд (трех при наличии полного набора логических команд) путем использования выражения для $x \overset{u}{\leq} y$ из раздела 2.11 с упрощениями, основанными на том, что в строке 1 известно, что $n_{31} = 1$, а в строке 5 — что $d_{31} = 0$. Это позволяет упростить выражения.

В строке 1: $n \overset{u}{\geq} d = \left(n \,\&\, \neg(n - d)\right) \overset{u}{\gg} 31$

В строке 5: $r \overset{u}{\geq} d = \left(r \mid \neg(r - d)\right) \overset{u}{\gg} 31$

Мы можем получить код без ветвлений, если считать, что при $d \overset{u}{\geq} 2^{31}$ частное должно быть равно 0. В этом случае делитель можно использовать в команде *знакового деления,* поскольку при неверном его распознавании как отрицательной величины результат устанавливается равным 0 (что находится в корректных переделах частного — до 1). Случай большого делимого обрабатывается, как и раньше, сдвигом перед делением на одну позицию вправо и сдвигом частного на одну позицию влево. Это дает нам следующую программу (требующую выполнения 10 базовых RISC-команд).

1. $t \leftarrow d \overset{s}{\gg} 31$
2. $n' \leftarrow n \,\&\, \neg t$
3. $q \leftarrow \left(\left(n' \overset{u}{\div} 2 \right) \div d \right) \times 2$
4. $r \leftarrow n - qd$
5. $q \leftarrow q + \left(r \overset{u}{\geq} d \right)$

9.4. Беззнаковое длинное деление

Под "длинным делением" подразумевается деление двойного слова на одинарное. В случае 32-битовой машины это деление $64 \overset{u}{\div} 32 \Rightarrow 32$ с неопределенным результатом в случае переполнения (в частности, при делении на 0).

Некоторые 32-битовые компьютеры оснащены командами беззнакового длинного деления, однако эти команды малоупотребимы в связи с тем, что из большинства языков программирования высокого уровня доступно только деление вида $32 \overset{u}{\div} 32 \Rightarrow 32$. Таким образом, разработчики компьютеров могут предпочесть обеспечить наличие только команды деления формата $32 \overset{u}{\div} 32$. Далее будут приведены два алгоритма, которые обеспечивают реализацию отсутствующего в компьютере беззнакового длинного деления.

Аппаратный алгоритм сдвига и вычитания

В качестве первой попытки выполнения длинного деления рассмотрим, как это деление реализуется аппаратно. Существует два широко распространенных алгоритма, называемых восстанавливающим (restoring) и невосстанавливающим (nonrestoring) делением [31, sec. A–2; 14]. Оба алгоритма представляют собой алгоритмы "сдвига и вычитания". В восстанавливающей версии, показанной ниже, восстанавливающий шаг состоит в прибавлении делителя, когда вычитание дает отрицательный результат. Величины x, y и z хранятся в 32-битовых регистрах. Изначально $x \| y$ представляет собой делимое размером в два слова, а делитель располагается в z. Для хранения переполнения при вычитании нам потребуется однобитовый регистр c. Вот как выглядит этот алгоритм.

```
do i←1 to 32
   c ‖ x ‖ y ← 2(x ‖ y)                        // Сдвиг влево на один бит
   c ‖ x ← (c ‖ x) - (0b0 ‖ z)                 // Вычитание(33 бита)
   y₀ ← ¬c                                     // Установка одного бита частного
   if c then c ‖ x ← (c ‖ x) + (0b0 ‖ z)       // Восстановление
end
```

По окончании вычислений частное находится в регистре y, а остаток — в регистре x.

В случае переполнения этот алгоритм не дает никаких полезных результатов. При делении величины $x \| y$ на 0 частное представляет собой поразрядное дополнение x до единиц, а остаток — величину y. В частности, $0 \div^{u} 0 \Rightarrow 2^{32} - 1 \text{ rem } 0$. Охарактеризовать прочие случаи переполнения гораздо сложнее.

Было бы неплохо, если бы при ненулевом делителе алгоритм давал корректное значение частного по модулю 2^{32} и корректное значение остатка. Однако, похоже, единственный способ добиться этого — создать регистр длиной свыше 97 бит для представления $c \| x \| y$ и выполнить тело цикла 64 раза, что обеспечит выполнение деления вида $62 \div^{u} 32 \Rightarrow 64$. Вычитание при этом остается 33-битовой операцией.

Реализовать этот алгоритм программно достаточно сложно, поскольку у большинства компьютеров нет 33-битового регистра, который бы хранил значение $c \| x$. В листинге 9.2 приведен вариант алгоритма, который в определенной степени следует рассмотренному аппаратному алгоритму.

Листинг 9.2. Беззнаковое длинное деление, алгоритм сдвига и вычитания

```
unsigned divlu(unsigned x, unsigned y, unsigned z)
{
   // Деление (x || y) на z
   int i;
   unsigned t;
   for (i = 1; i <= 32; i++)
   {
      t = (int)x >> 31;            // Все биты = 1,
                                   // если x(31) = 1
      x = (x << 1) | (y >> 31);    // Сдвиг x||y влево
      y = y << 1;                  // на один бит
      if ((x | t) >= z)
      {
         x = x - z;
```

```
            y = y + 1;
        }
  }
  return y;                         // Остаток хранится в x
}
```

Переменная t используется для того, чтобы обеспечить правильное сравнение. После сдвига x||y необходимо выполнить 33-битовое сравнение. Если первый бит x равен 1 (до выполнения сдвига), то совершенно очевидно, что интересующая нас 33-битовая величина больше, чем 32-битовый делитель. В этом случае все биты x|t равны единице, так что сравнение с z дает правильный результат (true). Если же первый бит x равен 0, то 32-битового сравнения достаточно.

Код в листинге 9.2 требует выполнения от 321 до 358 базовых RISC-команд, в зависимости от того, насколько часто результат сравнения оказывается истинен. Если компьютер имеет команду *сдвига влево двойного слова,* операция сдвига может быть выполнена с помощью одной команды вместо четырех. Это позволяет снизить общее количество выполняемых команд до 225–289 (в предположении, что для управления циклом требуется выполнение двух команд на итерацию).

Если принять x=0, то алгоритм из листинга 9.2 может быть использован для реализации деления $32 \overset{u}{\div} 32 \Rightarrow 32$. Единственное упрощение при этом состоит в том, что переменную t можно не использовать, поскольку ее значение всегда равно 0.

Ниже приведен невосстанавливающий аппаратный алгоритм беззнакового деления. Основная идея состоит в том, что после вычитания делителя z из 33-битовой величины $c \| x$ нет необходимости добавлять z, если результат оказался отрицательным. Вместо этого в следующей итерации достаточно выполнить *сложение* вместо *вычитания*. Это связано с тем, что добавление z (для коррекции ошибки, связанной с вычитанием z в предыдущей итерации), сдвиг влево и вычитание z эквивалентно его добавлению ($2(u+z) - z = 2u + z$). Преимущество данного метода в контексте аппаратной реализации состоит в том, что в каждой итерации цикла выполняется только одна команда сложения или вычитания и сумматор оказывается самой медленной схемой в цикле[4]. В конце алгоритма требуется корректировка остатка, если он отрицателен (соответствующая корректировка частного не требуется).

Делимое представляет собой двойное слово $x \| y$, а делитель — слово z. По окончании вычислений частное находится в регистре y, а остаток — в регистре x.

```
c = 0
do i ← 1 to 32
   if c = 0 then do
      c || x || y ← 2(x || y)          // Сдвиг влево на один бит
      c || x ← (c || x) − (0b0 || z)    // Вычитание делителя
   end
   else do
```

[4] В действительности восстанавливающий алгоритм деления может избежать восстанавливающего шага, помещая результат вычитания в отдельный регистр и записывая этот регистр в x, только если результат 33-битового вычитания неотрицателен. Однако в некоторых реализациях для этого требуется дополнительный регистр и, возможно, большее время работы.

$c \| x \| y \leftarrow 2(x \| y)$ // Сдвиг влево на один бит

$c \| x \leftarrow (c \| x) + (0b0 \| z)$ // Прибавление делителя

end

$y_0 \leftarrow \neg c$ // Установка одного бита частного

end

if $c = 1$ then $x \leftarrow x + z$ // Коррекция отрицательного остатка

Микрокомпьютер 801 (ранняя экспериментальная RISC-разработка IBM) имеет команду *пошагового деления,* которая, по сути, выполняет приведенные выше в теле цикла действия. Эта команда использует бит переноса для хранения *c* и MQ (32-биттовый регистр) для хранения *y*. Для реализации данной команды требуется 33-битовый сумматор и 32-битовое вычитающее устройство. Команда *пошагового деления* микрокомпьютера 801 несколько сложнее, чем действия, выполняемые в теле цикла, поскольку выполняет знаковое деление и оснащена проверкой переполнения. При ее использовании подпрограмма деления может быть записана с помощью 32 последовательных команд *пошагового деления,* за которыми следует коррекция частного и остатка (для того, чтобы обеспечить верный знак остатка).

Использование короткого деления

Алгоритм для деления $64 \overset{u}{\div} 32 \Rightarrow 32$ может быть получен из алгоритма беззнакового деления больших целых чисел в листинге 9.1 для частного случая $m=4$ и $n=2$. Кроме того, требуется внесение ряда других изменений. Параметры в данном случае должны представлять собой полные слова, передаваемые по значению, а не массивы полуслов. Условие переполнения также отличается от исходного; в данном случае переполнение происходит, если частное не помещается в полном слове. Все это дает возможность внести определенные упрощения в программу. Можно показать, что оценка qhat в данном случае всегда точная, так как делитель состоит только из двух цифр по полслова. Это означает, что корректирующий шаг с прибавлением делителя может быть опущен. Если развернуть "главный цикл" листинга 9.1 и цикл внутри него, становятся возможны еще некоторые небольшие упрощения кода.

Результат этих преобразований показан в листинге 9.3. Делимое находится в словах `u1` и `u0`; в ul содержится старшее слово. Параметр v содержит делитель. Функция по завершении вычислений возвращает значение частного. Если вызывающая функция передает в качестве `r` ненулевой указатель, то остаток будет возвращен в слове, на которое указывает `r`.

Листинг 9.3. Длинное беззнаковое деление с использованием команды деления полных слов

```
unsigned divlu(unsigned u1, unsigned u0, unsigned v,
               unsigned *r)
{
    const unsigned b
                = 65536;  // Основание счисления (16 битов)
    unsigned un1, un0,    // Слова нормализованного делимого
             vn1, vn0,    // Цифры нормализованного делителя
             q1, q0,      // Цифры частного
             un32, un21,  // Пары цифр делимого
             un10,
             rhat;        // Остаток
    int s;                // Величина сдвига нормализации
    if (ul >= v)             // При переполнении
```

```
    {                               // устанавливается невозможное
        if (r != NULL)             // значение остатка и
            *r = 0xFFFFFFFF;       // возвращается максимально
        return 0xFFFFFFFF;         // возможное число
    }
    s = nlz(v);                     // 0 <= s <= 31.
    v = v << s;                     // Нормализация делителя
    vn1 = v >> 16;                  // Разбиение делителя
    vn0 = v & 0xFFFF;               // на две 16-битовые цифры
    un32 = (u1 << s) | (u0 >> 32 - s) & (-s >> 31);
    un10 = u0 << s;                 // Сдвиг делимого влево
    un1 = un10 >> 16;               // Разбиение правой половины
    un0 = un10 & 0xFFFF;            // делимого на две цифры
    q1 = un32/vn1;                  // Вычисление первой цифры
    rhat = un32 - q1*vn1;           // частного q1
again1:
    if (q1 >= b || q1*vn0 > b*rhat + un1)
    {
        q1 = q1 - 1;
        rhat = rhat + vn1;
        if (rhat < b) goto again1;
    }
    un21 = un32*b + un1 - q1*v;// Умножение и вычитание
    q0 = un21/vn1;                  // Вычисление второй цифры
    rhat = un21 - q0*vn1;           // частного q0
again2:
    if (q0 >= b || q0*vn0 > b*rhat + un0)
    {
        q0 = q0 - 1;
        rhat = rhat + vn1;
        if (rhat < b) goto again2;
    }
    if (r != NULL)                  // Если требуется,
        *r = (un21*b + un0          // вычисляем остаток
              - q0*v) >> s;
    return q1*b + q0;
}
```

В качестве индикатора переполнения программа возвращает значение остатка, равное максимально возможному целому числу. Такой остаток невозможен ни при каком корректном делении, поскольку остаток всегда меньше делителя. Кроме того, при переполнении функция возвращает частное, равное максимально возможному целому числу (что также может служить индикатором переполнения в случае, когда вызывающая функция не запрашивает значение остатка).

Странное выражение `(-s>>31)` в присвоении `u32` предназначено для того, чтобы программа работала в случае s = 0 на машинах, сдвиг у которых выполняется по модулю 32 (например, в Intel x86).

Эксперименты с равномерно распределенными случайными числами показывают, что тело цикла `again` выполняется примерно около 0.38 раза на каждый вызов функции. Это дает нам среднее количество выполняемых команд, равное 52. Среди них одна команда подсчета *количества ведущих нулевых битов,* две — *деления,* а также 6.5 — умножения (не считая умножений на `b`, которые выполняются при помощи *сдвигов).* Если требуется значение остатка, добавляется шесть команд (включаю команду сохранения `r`), одна из которых — умножение.

Что касается знаковой версии divlu, то, вероятно, пошаговая модификация приведенного в листинге 9.3 кода для получения знакового варианта представляет собой непростую задачу. Однако данный алгоритм можно использовать для реализации знакового деления, если взять абсолютное значение аргументов, вызвать divlu и обратить знак результата, если знаки аргументов были различны. При этом не возникает никаких проблем с экстремальными значениями типа наибольшего по абсолютной величине отрицательного значения, так как абсолютное значение любого знакового числа корректно представимо беззнаковым числом. Такой алгоритм показан в листинге 9.4.

Листинг 9.4. Длинное знаковое деление с использованием длинного беззнакового деления

```
int divls(int u1, unsigned u0, int v, int *r)
{
    int q, uneg, vneg, diff, borrow;
    uneg = u1 >> 31; // -1, если u < 0.
    if (uneg)
    {                     // Вычисление абсолютного
        u0 = -u0;         // значения делимого и
        borrow = (u0 != 0);
        u1 = -u1 - borrow;
    }
    vneg = v >> 31;  // -1, если v < 0.
    v = (v ^vneg) - vneg; // Абсолютное значение v
    if ((unsigned)u1 >= (unsigned)v) goto overflow;
    q = divlu(u1, u0, v, (unsigned *)r);
    diff = uneg ^ vneg;    // Изменяем знак q, если
    q = (q ^diff) - diff; // знаки u и v различны
    if (uneg && r != NULL)
        *r = -*r;
    if ((diff ^q) < 0 && q != 0)
    {                            // При переполнении даем
    overflow:                    // остатку невозможное
        if (r != NULL)           // значение и возвращаем
            *r = 0x80000000;  // отрицательное частное
        q = 0x80000000;          // с максимально возможным
    }                            // абсолютным значением
    return q;
}
```

В случае знакового деления достаточно трудно разработать действительно хороший код для обнаружения переполнения. Алгоритм, приведенный в листинге 9.4, выполняет предварительное обнаружение переполнения, идентичное случаю длинного беззнакового деления, которое гарантирует, что $|u/v| < 2^{32}$. После этого нам остается только убедиться, что остаток имеет корректный знак или равен 0.

ГЛАВА 10

ЦЕЛОЕ ДЕЛЕНИЕ НА КОНСТАНТЫ

На многих компьютерах операция деления выполняется довольно медленно, и по возможности ее использования стараются избежать. Время выполнения команды деления, как правило, превышает время выполнения сложения в 20 и более раз, причем обычно оно одинаково велико как для больших, так и для малых операндов. В этой главе приводится ряд методов, позволяющих избежать команды деления в случаях, когда делитель представляет собой константу.

10.1. Знаковое деление на известную степень 2

Многие ошибочно считают, что *знаковый сдвиг вправо на k позиций* делит число на 2^k с использованием обычного отекающего деления [20]. Однако на самом деле не все так просто. Приведенный ниже код выполняет деление $q = n \div 2^k$, где $1 \le k < 31$ [29].

```
shrsi  t,n,k-1        Формируем целое число
shri   t,t,32-k      2**k - 1, если n < 0; иначе 0
add    t,n,t          Добавляем его к п
shrsi  q,t,k          и выполняем знаковый сдвиг
вправо
```

Этот код не содержит команд ветвления. Он может быть упрощен до трех команд при делении на 2 *(k=* 1). Однако данный код имеет смысл только на компьютерах, которые способны выполнить сдвиг на большое значение за малое время. Случай $k=31$ не имеет особого смысла, поскольку число 2^{31} не представимо на компьютере. Тем не менее приведенный код дает правильный результат и в этом случае ($q=-1$, если я $= -2^{31}$, и $q=0$ для прочих значений *n).*

Для деления на -2* после приведенного выше кода должна следовать команда изменения знака. Лучшего варианта для данного деления пока не придумано.

Вот еще один, более очевидный и простой код для деления на 2^k:

```
        bge    n,label     Ветвление при п >= O
        addi   n,n,2**k-1  Прибавление 2**k - 1 к n
label   shrsi  n,n,k       и знаковый сдвиг вправо
```

Этот код предпочтительнее использовать на машине с медленным сдвигом и быстрыми командами ветвления.

Компьютер PowerPC имеет необычное устройство для ускорения деления на степень 2 [16]. Команда *знакового сдвига вправо* устанавливает бит переноса, если сдвигаемое число отрицательно и был сдвиг одного или нескольких единичных битов. Кроме того, данный компьютер имеет команду addze для сложения бита переноса с регистром. Все это позволяет реализовать деление на любую (положительную) степень 2 с помощью двух команд:

```
shrsi  q,n,k
addze  q,q
```

Единственная команда сдвига shrsi на *k* позиций выполняет знаковое деление на 2^k, которое совпадает как с модульным делением, так и с делением с округлением к мень-

шему значению. Это наводит на мысль о том, что такое деление может быть предпочтительнее для использования в языках высокого уровня, чем деление с отсечением, так как позволяет компилятору для деления на 2 использовать единственную команду shrsi. Кроме того, команда shrsi с последующей за ней командой neg выполняет модульное деление на -2^k, что, в свою очередь, указывает на преимущества использования модульного деления (впрочем, в основном это вопрос эстетического восприятия, в силу того что задача деления на отрицательную константу встречается достаточно редко).

10.2. Знаковый остаток от деления на степень 2

Если нам требуется вычислить и частное и остаток от деления я $\div 2^k$, простейший путь получения значения остатка — вычислить его по формуле $r = \partial * 2^* - n$, для чего после вычисления частного требуется выполнение двух команд:

```
shli  r,q,k
sub   r,r,n
```

Для вычисления одного лишь остатка необходимо выполнить около четырех или пяти команд. Один из способов вычисления состоит в использовании рассмотренной ранее последовательности из четырех команд для знакового деления на 2^k, за которой следуют две команды вычисления остатка. При этом две последовательные команды *сдвига* можно заменить командой *и,* что дает нам решение из пяти команд (четырех при $k=1$).

```
shrsi  t,n,k-1     Формируем целое число
shri   t,t,32-k    2**k - 1, если n < 0; иначе 0
add    t,n,t       Добавляем его к n,
andi   t,t,-2**k   сбрасываем k правых битов
sub    r,n,t       и вычитаем его из п
```

Еще один метод основан на том, что

$$\mathrm{rem}(n,2^k) = \begin{cases} n \& (2^k - 1), & n \ge 0, \\ -((-n) \& (2^k - 1)), & n < 0. \end{cases}$$

Для того чтобы воспользоваться этой формулой, сначала вычисляем $t \leftarrow n \overset{s}{\gg} 31$, а затем

$$r \leftarrow ((\mathrm{abs}(n) \& (2^k - 1)) \oplus t) - t$$

(требуется пять команд) или в случае $k=1$, так как $(-n) \& 1 = n \& 1$, вычисляем

$$r \leftarrow ((n \& 1) \oplus t) - t$$

(требуется четыре команды). Этот метод не очень хорош при $k>1$, если компьютер не оснащен командой вычисления *абсолютного значения* (в таком случае вычисление остатка будет требовать выполнения семи команд).

Еще один метод основан на том, что

$$\mathrm{rem}(n,2) = \begin{cases} n \& (2^k - 1), & n \ge 0, \\ ((n + 2^k - 1) \& (2^k - 1)) - (2^k - 1), & n < 0. \end{cases}$$

Эта формула приводит к вычислениям

$$t \leftarrow \left(n \overset{s}{\gg} k-1\right) \overset{u}{\gg} 32-k,$$
$$r \leftarrow \left(\left(n+t\right) \& \left(2^k-1\right)\right)-t$$

(при $k>1$ требуется выполнение пяти команд, при $k=1$ — четырех).

Все перечисленные методы работают при $1 < k < 31$.

Кстати, если в компьютере нет команды *знакового сдвига вправо,* то значение, которое равно 2^k-1 для $n<0$ и 0 для $n>0$, может быть построено следующим образом:

$$t_1 \leftarrow n \overset{u}{\gg} 31,$$
$$r \leftarrow (t_1 \ll k)-t_1,$$

что приводит к добавлению только одной команды.

10.3. Знаковое деление и вычисление остатка для других случаев

Основной прием в этом случае состоит в умножении на некоторое соответствующее делителю d число, примерно равное $2^{32}/d$, с последующим выделением 32 левых битов произведения. Однако реальные вычисления для ряда делителей, в частности для 7, оказываются существенно сложнее.

Рассмотрим сначала несколько конкретных примеров, которые пояснят код, генерируемый обобщенным методом. Обозначим регистры следующим образом:

n — входное целое число (числитель);
M — в него загружается "магическое число";
t — временный регистр;
q — в нем будет размещено частное;
r — в нем будет размещен остаток.

Деление на 3

```
li    M,0x55555556  Загрузка "магического числа"  (2**32+2)/3
mulhs q,M,n         q = floor(M*n/2**32)
shri  t,n,31        Прибавляем 1 к q, если
add   q,q,t         n отрицательно

muli  t,q,3         Вычисляем остаток как
sub   r,n,t         r = n - q*3
```

Доказательство. Операция *старшее слово знакового умножения* не может вызвать переполнения, поскольку произведение двух 32-битовых чисел всегда может быть представлено 64-битовым числом, а команда mulhs возвращает старшие 32 бита 64-битового произведения. Это действие эквивалентно делению 64-битового произведения на 2^{32} и вычислению функции floor() от полученного результата, причем это замечание справедливо независимо от того, отрицательно рассматриваемое произведение или положительно. Таким образом, при $n \ge 0$ приведенный выше код вычисляет

$$q = \left\lfloor \frac{2^{32}+2}{3}\frac{n}{2^{32}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{2n}{3 \cdot 2^{32}} \right\rfloor.$$

Далее, $n < 2^{31}$, поскольку $2^{31}-1$ является наибольшим **представимым** целым числом. Следовательно, член-ошибка $2n/(3\cdot 2^{32})$ меньше 1/3 (и это значение неотрицательно), так что в соответствии с теоремой D4 из раздела 9.1 получаем $q = \lfloor n/3 \rfloor$, что и требовалось получить (формула (1) в том же разделе 9.1).

В случае $n < 0$ к частному добавляется 1. Следовательно, приведенный выше код вычисляет в этом случае величину

$$q = \left\lfloor \frac{2^{32}+2}{3} \frac{n}{2^{32}} \right\rfloor + 1 = \left\lfloor \frac{2^{32}n + 2n + 3\cdot 2^{32}}{3\cdot 2^{32}} \right\rfloor = \left\lceil \frac{2^{32}n + 2n + 1}{3\cdot 2^{32}} \right\rceil .$$

Здесь была использована теорема D2. Следовательно,

$$q = \left\lceil \frac{n}{3} + \frac{2n+1}{3\cdot 2^{32}} \right\rceil .$$

При $-2^{31} < n < -1$ получаем

$$-\frac{1}{3} + \frac{1}{3\cdot 2^{32}} \le \frac{2n+1}{3\cdot 2^{32}} \le -\frac{1}{3\cdot 2^{32}} .$$

Таким образом, ошибка отрицательна и больше, чем $-1/3$, а значит, в соответствии с теоремой D4 $q = \lceil n/3 \rceil$, что и является искомым результатом (формула (1) из раздела 9.1).

Итак, установлено, что вычисленное значение частного корректно. Тогда корректно и значение остатка, поскольку остаток должен удовлетворять соотношению $n = qd + r$. Умножение на 3 не может вызвать переполнения (так как $-2^{31}/3 \le q \le (2^{31}-1)/3$), а *вычитание* не может привести к переполнению в связи с тем, что результат должен находиться в диапазоне от -2 до +2.

Команда *умножения на непосредственно заданное значение* может быть выполнена с помощью двух *сложений* или *сдвига* и *сложения,* в том случае, если такая замена дает выигрыш во времени вычислений.

На многих современных RISC-компьютерах частное может быть вычислено так, как показано выше, за девять-десять тактов, в то время как команда *деления* может потребовать 20 тактов или около того.

Деление на 5

Для деления на 5 хотелось бы использовать код того же **типа,** что и для деления на 3, но, понятно, с множителем $(2^{32}+4)/5$. К сожалению, при этом ошибка оказывается слишком большой и результат отличается на единицу от точного примерно для пятой части значений $|n| \ge 2^{30}$. Но оказывается, можно использовать множитель $(2^{33}+3)/5$ и добавить к коду команду *знакового сдвига вправо.* В результате получается следующий код:

```
li    M,0x66666667  Загрузка "магического числа"  (2**33+3)/5
mulhs q,M,n         q = floor(M*n/2**32)
shrsi q,q,1
shri  t,n,31        Прибавляем 1 к q, если
add   q,q,t          n отрицательно

muli  t,q,5         Вычисляем остаток как
sub   r,n,t          r = n - q*5
```

Доказательство. Команда `mulhs` дает 32 старших бита 64-битового произведения, после чего код знаково сдвигает полученное значение вправо на одну позицию. Это действие эквивалентно делению 64-битового произведения на 2^{32} и вычислению функции floor() от полученного результата. Таким образом, для $n \geq 0$ приведенный код вычисляет

$$q = \left\lfloor \frac{2^{33}+3}{5} \frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{5} + \frac{3n}{5 \cdot 2^{33}} \right\rfloor.$$

Для $0 \leq n < 2^{31}$ ошибка составляет $3n/(5 \cdot 2^{33})$; это значение неотрицательно и меньше, чем $1/5$, так что в соответствии с теоремой D4 $q = \lfloor n/5 \rfloor$.

При $n < 0$ приведенный выше код вычисляет значение

$$q = \left\lfloor \frac{2^{33}+3}{5} \frac{n}{2^{33}} \right\rfloor + 1 = \left\lceil \frac{n}{5} + \frac{3n+1}{5 \cdot 2^{33}} \right\rceil.$$

Здесь член-ошибка отрицателен и превышает $-1/5$, так что $q = \lceil n/5 \rceil$. Корректность вычисленного остатка доказывается так же, как и в случае деления на *3*. Как и ранее, команда *умножения на непосредственно заданное значение* может быть заменена — в данном случае *сдвигом влево* на две позиции и *сложением*.

Деление на 7

При делении на 7 возникают новые проблемы. Множители $(2^{32}+3)/7$ и $(2^{33}+6)/7$ дают слишком большие значения ошибки. Множитель $(2^{34}+5)/7$ подходит, но он слишком велик для размещения в 32-битовом знаковом слове. Однако умножение на такое большое число можно выполнить путем умножения на отрицательное число $(2^{34}+5)/7 - 2^{32}$ с последующей коррекцией произведения путем сложения. В результате получаем следующий код для деления на 7:

```
li     M,0x92492493   "Магическое число" (2**34+5)/7 - 2**32
mulhs  q,M,n          q = floor(M*n/2**32).
add    q,q,n          q = floor(M*n/2**32) + n
shrsi  q,q,2          q = floor(q/4)
shri   t,n,31         Прибавляем 1 к q, если
add    q,q,t          n отрицательно

muli   t,q,7          Вычисляем остаток как
sub    r,n,t          r = n - q*7
```

Доказательство. Важно обратить внимание на то, что команда "add `q,q,n`" не может вызвать переполнения. Дело в том, что q и n имеют противоположные знаки; это связано с умножением на отрицательное число. Таким образом, это "компьютерное сложение" выполняется так же, как и обычное арифметическое сложение, и для $n \geq 0$ приведенный выше код вычисляет

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) \Big/ 4 \right\rfloor = \left\lfloor \left\lfloor \frac{2^{34}n + 5n - 7 \cdot 2^{32}n + 7 \cdot 2^{32}n}{7 \cdot 2^{32}} \right\rfloor \Big/ 4 \right\rfloor = \left\lfloor \frac{n}{7} + \frac{5n}{7 \cdot 2^{34}} \right\rfloor$$

(здесь при преобразованиях использовано следствие из теоремы D3).

Для $0 \le n < 2^{31}$ ошибка, составляющая $5n/(7 \cdot 2^{34})$, неотрицательна и меньше $1/7$, так что $q = \lfloor n/7 \rfloor$.

Для л < 0 приведенный выше код вычисляет значение

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) \Big/ 4 \right\rfloor + 1 = \left\lceil \frac{n}{7} + \frac{5n+1}{7 \cdot 2^{34}} \right\rceil.$$

В этом случае ошибка не положительна и больше -1/7 , так что $q = \lceil n/7 \rceil$.

Команда *умножения на непосредственно заданное значение* может быть заменена *сдвигом влево* на три позиции и *вычитанием*.

10.4. Знаковое деление на делитель, не меньший 2

После рассмотренного материала вы можете заинтересоваться, не возникают ли новые проблемы при работе с другими делителями. В этом разделе вы узнаете, что не возникают: все проблемы при делителе $d > 2$ исчерпываются тремя рассмотренными случаями.

Некоторые из приводимых доказательств весьма сложны, так что читайте дальнейший материал внимательно и не забывайте о том, что работаете со словом размером W.

Итак, пусть дан размер слова $W \ge 3$ и делитель $2 \le d < 2^{W-1}$ и требуется найти наименьшее целое $0 < m < 2^W$ и целое $p > W$, такие, что

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \text{ для } 0 \le n < 2^{W-1} \text{ и} \tag{1,а}$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \text{ доя } -2^{W-1} \le n \le -1. \tag{1,б}$$

Найти *наименьшее* целое m необходимо потому, что меньший множитель может потребовать меньшую величину сдвига (возможно, 0) либо привести к коду наподобие кода из примера "деление на 5", но не "деление на 7". Требование $m < 2^W - 1$ обеспечивает не большее количество команд, чем в примере деления на 7 (т.е. с множителем в диапазоне от 2^{W-1} до $2^W - 1$ можно справиться с помощью дополнительной команды сложения, как это было сделано в примере деления на 7, но предпочтительнее обойтись меньшими множителями). Требование $p > W$ нужно постольку, поскольку генерируемый код выделяет левую половину произведения *mn,* что эквивалентно сдвигу вправо на W позиций. Таким образом, общий сдвиг вправо составляет не менее W позиций.

Есть определенное различие между множителем m и "магическим числом", обозначаемым как M. Магическое число — это значение, которое используется в команде *умножения* и задается следующим образом:

$$M = \begin{cases} m, & 0 \le m < 2^{W-1}, \\ m - 2^W, & 2^{W-1} \le m < 2^W. \end{cases}$$

В силу того, что соотношение (1,б) должно выполняться при $n = -d$, т.е. $\lfloor -md/2^p \rfloor + 1 = -1$, получаем

$$\frac{md}{2^p} > 1. \tag{2}$$

Пусть n_c — наибольшее (положительное) значение n, такое, что $\text{rem}(n_c,d)=d-1$. Оно существует, поскольку имеется, как минимум, одно возможное значение $n_c = d - 1$. Его можно вычислить как

$$n_c = \lfloor 2^{W-1}/d \rfloor d - 1 = 2^{W-1} - \text{rem}(2^{W-1},d) - 1\,.$$

Поскольку n_c представляет собой одно из d наибольших допустимых значений л, то

$$2^{W-1} - d \le n_c \le 2^{W-1} - 1 \qquad (3,a)$$

и, очевидно,

$$n_c \ge d - 1\,. \qquad (3,б)$$

Поскольку (1,а) должно выполняться при n - n_c,

$$\left\lfloor \frac{mn_c}{\text{ОТ}} \right\rfloor = \left\lfloor \frac{n_c}{\text{Та}} \right\rfloor = \frac{n_c-(d-1)}{d}$$

или

$$\frac{mn_c}{2^p} < \frac{n_c+1}{d}\,.$$

Объединяя полученный результат с (2), получим

$$\frac{2''}{d} < m < \frac{2^p}{d}\frac{n_c+1}{n_c}\,. \qquad (4)$$

Поскольку m должно быть наименьшим целым, удовлетворяющим (4), оно представляет собой целое число, следующее за $2^p/d$, т.е.

$$\boxed{m = \frac{2^p + d - \text{rem}(2^p,d)}{d}}\,. \qquad (5)$$

Комбинируя полученную формулу с правой частью (4), получим:

$$\boxed{2^p > n_c\left(d - \text{rem}(2^p,d)\right)}\,. \qquad (6)$$

Алгоритм

Таким образом, алгоритм поиска магического числа M и величины сдвига s для данного делителя d начинается с вычисления n_c, а затем неравенство (6) решается путем подстановки последовательных возрастающих значений. Если $p < W$, то устанавливаем $p = W$ (теорема ниже показывает, что данное значение также удовлетворяет неравенству (6)). Когда найдено наименьшее $p > W$, удовлетворяющее неравенству (6), из (5) вычисляется значение m. Это наименьшее возможное значение m, поскольку нами найдено наименьшее приемлемое p, а из (4) понятно, что, чем меньше значение p, тем меньше и значение m. И наконец, $s = p$ - W к M просто являются интерпретацией m как знакового целого числа (как оно рассматривается командой `mulhs`).

То, что при $p < W$ мы устанавливаем $p = W$, обосновывается следующей теоремой.

Теорема DC1. *Если для некоторого значения p справедливо неравенство (6), то оно справедливо и для больших значений p.*

Доказательство. Предположим, что неравенство (6) справедливо для $p = p_0$. Умножение (6) на 2 дает

$$2^{p_0+1} > n_c\left(2d - 2\,\mathrm{rem}\left(2^{p_0}, d\right)\right).$$

Из теоремы D5 $\mathrm{rem}\left(2^{p_0+1}, d\right) \ge 2\,\mathrm{rem}\left(2^{p_0}, d\right) - d$. Объединяя эти выражения, получим

$$2^{p_0+1} > n_c\left(2d - \left(\mathrm{rem}\left(2^{p_0+1}, d\right) + d\right)\right) \text{ или } 2^{p_0+1} > n_c\left(d - \mathrm{rem}\left(2^{p_0+1}, d\right)\right)$$

Следовательно, неравенство (6) справедливо для $p = p_0 + 1$, а потому и для всех больших значений.

Таким образом, можно решить (6) путем бинарного поиска, хотя, по-видимому, предпочтительнее простой линейный поиск, начинающийся со значения $p = W$, поскольку d обычно мало, а малые значения d приводят к малым значениям p.

Доказательство пригодности алгоритма

Покажем, что (6) всегда имеет решение и что $0 < m < 2^W$ (нет необходимости показывать, что $p \ge W$, так как это условие выполняется принудительно).

Покажем, что (6) всегда имеет решение, получив верхнюю границу p. Кроме того, в познавательных целях получим также нижнюю границу в предположении, что p не обязано быть равным как минимум W. Для получения указанных границ p заметим, что для любого положительного целого x существует степень 2, большая x и не превосходящая $2x$. Следовательно, из (6):

$$n_c\left(d - \mathrm{rem}\left(2^p, d\right)\right) < 2^p \le 2n_c\left(d - \mathrm{rem}\left(2^p, d\right)\right).$$

Поскольку $0 \le \mathrm{rem}\left(2^p, d\right) \le d - 1$,

$$n_c + 1 \le 2^p \le 2n_c d. \qquad (7)$$

Из (3,а) и (3,б) получаем $n_c \ge \max\left(2^{W-1} - d, d - 1\right)$. Графики функций $f_1(d) = 2^{W-1} - d$ и $f_2(d) = d - 1$ в точке $d = \left(2^{W-1} + 1\right)/2$. Следовательно, $n_c > \left(2^{W-1} - 1\right)/2$. Поскольку n_c — целое число, $n_c \ge 2^{W-2}$. С учетом того, что $n_c, d \le 2^{W-1} - 1$, (7) превращается в

$$2^{W-2} + 1 \le 2^p \le 2\left(2^{W-1} - 1\right)^2$$

или

$$W - 1 \le p \le 2W - 2. \qquad (8)$$

Нижняя граница $p = W - 1$ вполне может быть достигнута (например, для $W=32$, $d=3$), но в этом случае мы устанавливаем $p = W$.

Если не делать p равным W принудительно, то из (4) и (7) получаем

$$\frac{n_c + 1}{d} < m < \frac{2n_c d}{d}\frac{n_c + 1}{n_c}.$$

Применение (3,б) дает

$$\frac{d - 1 + 1}{d} < m < 2\left(n_c + 1\right).$$

Так как $n_c \le 2^{W-1}-1$ (3,а),

$$2 \le m \le 2^W - 1 .$$

Если p принудительно делается равным W, то из (4) получаем

$$\frac{2^W}{d} < m < \frac{2^W}{d}\frac{n_c+1}{n_c}.$$

Поскольку $2 < d < 2^{W-1}-1$ и $n_c \ge 2^{W-2}$,

$$\frac{2^W}{2^{W-1}-1} < m < \frac{2^W}{2}\frac{2^{W-2}+1}{2^{W-2}} \text{ или}$$

$$3 \le m \le 2^{W-1}+1 .$$

Следовательно, в любом случае m находится в пределах границ для схемы, проиллюстрированной примером деления на 7.

Доказательство корректности произведения

Покажем, что если p и m вычисляются из (6) и (5), то выполняются уравнения (1,а) и (1,б).

Уравнение (5) и неравенство (6), как легко видеть, имеют следствием (4). (В случае, когда p принудительно приравнивается к W, как показывает теорема DC1, неравенство (6) остается справедливым.) Далее рассмотрим отдельно пять диапазонов значений n:

$$0 < n < n_c,$$

$$n_c + 1 \le n \le n_c + d - 1,$$

$$-n_c \le n \le -1,$$

$$-n_c - d + 1 \le n \le -n_c - 1 \text{ и}$$

$$n = -n_c - d .$$

Из (4), поскольку m — целое число, следует:

$$\frac{2^p}{d} < m \le \frac{2^p(n_c+1)-1}{dn_c} .$$

При умножении на $n/2^p$ для $n > 0$ это соотношение превращается в

$$\frac{n}{d} < \frac{mn}{2^p} \le \frac{2^p n(n_c+1)-n}{2^p dn_c}, \text{ так что}$$

$$\left\lfloor \frac{n}{d} \right\rfloor < \left\lfloor \frac{mn}{2^p} \right\rfloor \le \left\lfloor \frac{n}{d} + \frac{(2^p-1)n}{2^p dn_c} \right\rfloor .$$

Если $0 < n < n_c$, то $0 \le (2^p-1)n/(2^p dn_c) < 1/d$, так что по теореме D4

$$\left\lfloor \frac{n}{d} + \frac{(2^p-1)n}{2^p dn_c} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor .$$

Следовательно, в случае $0 < n < n_c$ уравнение (1,а) выполняется.

Для л $> n_c$ значение n ограничено диапазоном

$$n_c \quad + \quad 1 \le n \le n_c + d - 1, \tag{9}$$

так как $n \ge n_c + d$ противоречит выбору n_c как наибольшего значения и, такого, что $\mathrm{rem}(n_c, d) = d - 1$ (кроме того, из (3,а) видно, что $n \ge n_c + d$ влечет за собой $n \ge 2^{W-1}$). Из (4), для $n > 0$, следует:

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n}{d}\frac{n_c + 1}{n_c}.$$

Путем элементарных алгебраических преобразований это неравенство может быть записано следующим образом:

$$\frac{n}{d} < \frac{mn}{2^p} \quad \frac{n_c+1}{d} \;, \; \frac{(n-n_c)(n_c+1)}{dn_c}. \tag{10}$$

Из (9) $1 \le n - n_c \le d - 1$, так что

$$0 \text{ - } \frac{(n-n_c)(n_c+1)}{dn_c} \le \frac{d-1}{d}\frac{n_c+1}{n_c}.$$

Поскольку (из (3,б)) $n_c \ge d - 1$ и $(n_c+1)/n_c$ максимально при минимальном n_c,

$$0 < \frac{(n-n_c)(n_c+1)}{dn_c} \le \frac{d-1}{d}\frac{d-1+1}{d-1} = 1.$$

В (10) член $(n_c+1)/d$ является целым числом. Член $(n-n_c)(n_c+1)/dn_c$ меньше или равен 1. Следовательно, (10) становится

$$\left\lfloor \frac{n}{d} \right\rfloor \le \left\lfloor \frac{mn}{2^p} \right\rfloor \le \frac{n_c+1}{d}.$$

Для всех n из диапазона (9) $\lfloor n/a \rfloor = (n_c + l)/d$. Следовательно, (1,а) выполняется и в этом случае ($n_c + 1 \le n \le n_c + d - 1$).

Для $n < 0$ из (4), так как m — целое число, имеем

$$\frac{2^p+1}{d} \le m < \frac{2^p}{d}\frac{n_c+1}{n_c}.$$

При умножении на $n/2^p$ с учетом того, что га < 0, это выражение преобразуется:

$$\frac{л}{d}\frac{n_c+1}{n_c} < \frac{mn}{2^p} \ge \frac{n}{d}\frac{2^p+1}{2^p}$$

или

$$\left\lfloor \frac{n}{d}\frac{n_c+1}{n_c} \right\rfloor + 1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lfloor \frac{n}{d}\frac{2^p+1}{2^p} \right\rfloor + 1.$$

Применив теорему D2, получим:

$$\left\lceil \frac{n_c(n_c+1) - dn_c + 1}{dn_c} \right\rceil + 1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n(2^p+1) - 2^p d + 1}{2^p d} \right\rceil + 1,$$

$$\left\lceil \frac{n(n_c+1)+1}{dn,} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n(2^p+1)+1}{2^p d} \right\rceil .$$

Поскольку $л + 1 < 0$, правое неравенство может быть ослаблено, что дает нам

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil . \qquad (11)$$

Для $-n_c < n < -1$

$$\frac{-n_c+1}{dn_c} < \frac{n+1}{dn_c} \le 0 \text{ или}$$

$$-\frac{1}{d} < \frac{n+1}{dn_c} \le 0 .$$

Следовательно, в силу теоремы D4

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil = \left\lceil \frac{n}{d} \right\rceil ,$$

так что в данном случае ($-n_c < n < -1$) выполняется уравнение (1,б).

При и $< -n_c$ величина n ограничена диапазоном

$$-n_c - d \le n \le -n_c - 1 . \qquad (12)$$

(Исходя из (3,а), из $n < -n_c - d$ вытекает $n < -2^{W-1}$, что невозможно.) Выполняя элементарные алгебраические преобразования в левой части (11), получаем

$$\left\lceil \frac{-n_c-1}{d} + \frac{(n+n_c)(n_c+1)+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d\backslash} \right\rceil . \qquad (13)$$

Для $-n_c - d \le n \le -n_c - 1$

$$\frac{(-d+1)(n_c+1)}{\text{rfn,.}} + \frac{1}{\text{rfn}_c} \le \frac{(n+n_c)(n_c+1)+1}{dn_c} \le \frac{-(n_c+1)+1}{dn_c} = -\frac{1}{d} .$$

Отношение $(n_c+1)/n_c$ максимально при минимальном значении n_c; таким образом, $n_c = d - 1$. Следовательно,

$$\frac{(-d+1)(d-1+1)}{d(d-1)} + \frac{1}{dn_c} \le \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0 \quad \text{или}$$

$$-1 \le \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0 .$$

Из (13), поскольку $(-n_c-1)/d$ — целое число, а прибавляемая величина находится между О и -1, получим:

$$\frac{-n_c-1}{d} \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil .$$

Для л из диапазона $-n_c - cl + 1 \le n \le -n_c - 1$

$$\left\lceil \frac{n}{d} \right\rceil = \frac{-я,-1}{d}.$$

Следовательно, $\lfloor mn/2Ј + 1 = \lceil n/d \rceil$, т.е. выполняется (1,6).

Последний случай, $п = -n_c - d$, может осуществиться только для некоторых значений d. Из (3,а) вытекает, что $-n_c - d \le -2^{W-1}$, поэтому если n принимает указанное значение, то $n = -n_c - d = -2^{W-1}$ и, следовательно, $n_c = 2^{W-1} - d$. Таким образом, $\text{rem}(2^{W-1}, d) = \text{rem}(n_c + d, d) = d\text{-}l$ (т.е. d **является** делителем $2^{W-1} + 1$).

Для рассматриваемого случая $n = -n_c - d$ формула (6) имеет решение $p = W - 1$ (наименьшее возможное значение p), поскольку при этом

$$\begin{aligned} n_c\left(d - \text{rem}(2^p, d)\right) &= \left(2^{W-1} - d\right)\left(d - \text{rem}(2^{W-1}, d)\right) = \\ &= \left(2^{W-1} - d\right)\left(d - (d-1)\right) = \\ &= 2^{W-1} - d < 2^{W-1} = 2^p. \end{aligned}$$

Тогда из (5)

$$m = \frac{2^{W-1} + d - \text{rem}(2^{W-1}, d)}{d} = \frac{2^{W-1} + d - (d-1)}{d} = \frac{2^{W-1} + 1}{d}.$$

Таким образом,

$$\begin{aligned} \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 &= \left\lfloor \frac{2^{W-1}+1}{d} \frac{-2^{W-1}}{2^{W-1}} \right\rfloor + 1 = \left\lfloor \frac{-2^{W-1}-1}{d} \right\rfloor + 1 = \\ &= \left\lceil \frac{-2^{W-1}-d}{d} \right\rceil + 1 = \left\lceil \frac{-2^{W-1}}{d} \right\rceil = \left\lceil \frac{n}{\backslash d} \right\rceil, \end{aligned}$$

так что уравнение (1,6) справедливо.

Этим завершается доказательство того, что если m $к$ p вычислены по формулам (5) и (6), то уравнения (1,а) и (1,б) выполняются для всех возможных значений n.

10.5. Знаковое деление на делитель, не превышающий -2

Поскольку знаковое целое деление удовлетворяет соотношению $n \div (-d) = -(n \div d)$, вполне приемлемой реализацией будет генерация кода для деления $n \div \backslash d \backslash$ с последующей командой обращения знака частного. (В этом случае мы не получим верный результат при делении на $d = -2^{W-1}$, но для этой и других отрицательных степеней 2 можно воспользоваться кодом из раздела 10.1, после которого просто изменить знак полученного результата.) Менять знак делимого при этом не следует из-за того, что делимое может оказаться максимальным по абсолютному значению отрицательным числом.

Однако можно избежать команды обращения знака. Схема такого вычисления имеет следующий вид:

$$q = \left\lfloor \frac{mn}{2^p} \right\rfloor \qquad \text{при } n < 0;$$

$$q = \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \qquad \text{при } n > 0.$$

Однако добавление 1 при положительных значениях и неудобно (так как невозможно просто использовать знаковый бит *n),* так что вместо этого будет выполняться прибавление 1, если значение q отрицательно. Эти действия эквивалентны, поскольку сомножитель m отрицателен (как будет показано далее).

Генерируемый для случая $W = 32$, $d = -7$ код показан ниже.

```
li     M,0x6DB6DB6D  Магическое число -(2**34+5)/7 + 2**32
mulhs  q,M,n         q = floor(M*n/2**32)
sub    q,q,n         q = floor(M*n/2**32) - n
shrsi  q,q,2         q = floor(q/4)
shri   t,q,31        Прибавляем 1 к q, если
add    q,q/t         q отрицательно (п положительно)

muli   t,q,-7        Вычисляем остаток по формуле
sub    r,n,t         r = n - q*(-7).
```

Этот код очень напоминает код для деления на +7, с тем отличием, что он использует множитель для деления на +7, но с обратным знаком, команду вычитания вместо команды умножения, а кроме того, команда сдвига shri использует *q,* а не *n* (о чем говорилось ранее). (Заметим, что в случае деления на +7 также можно использовать в качестве операнда *q,* но при этом код будет иметь меньшую степень параллелизма.) Команда *вычитания* не может привести к переполнению, поскольку операнды имеют один и тот же знак.

Тем не менее эта схема работает не всегда. Хотя приведенный код для $W = 32$, $d = -l$ вполне корректен, аналогичные изменения кода деления на 3 для получения кода для деления на -3 приведут к неверному результату при $W = 32$, $n = -2^{31}$.

Рассмотрим ситуацию подробнее.

Пусть дан размер слова $W > 3$ и делитель $-2^{W-1} \le d < -2$ и требуется найти наименьшее (по абсолютному значению) целое $-2^W < m < 0$ и целое $p > W$, такие, что

$$\left\lfloor \frac{mn}{2^p} \right\rfloor \quad \left\lfloor \frac{n}{d} \right\rfloor \quad \text{для } -2^{W-1} \le n \le 0, \tag{14,а}$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + \quad = \left\lceil \frac{n}{d} \right\rceil \quad \text{для } 1 \le n < 2^{W-1}. \tag{14,б}$$

Поступим так же, как и в случае деления на положительный делитель. Пусть n_c — наибольшее по абсолютной величине отрицательное значение *n,* такое, что $n_c = kd + 1$ для некоторого целого *k.* Такая величина n_c существует в силу того, что имеется по крайней мере одно значение $n_c = d + 1$. Это значение можно вычислить, исходя из того, что $n_c = \lfloor (-2^{W-1} - 1)/d \rfloor d + 1 = -2^{W-1} + \text{rem}(2^{W-1} + 1, d)$. Величина n_c представляет собой одно из $|d|$ наименьших допустимых значений *n,* так что

$$-2^{W-1} \le n_c \le -2^{W-1} - d - 1 \tag{15,а}$$

и, очевидно,

$$n_c \le d + 1. \tag{15,б}$$

Поскольку (14,б) должно выполняться при $n = -d$, а (14,а) — для $n = n_c$, то аналогично (4) получим

$$\frac{2^p}{d}\frac{n_c-1}{n_c} \quad \frac{2^p}{d} \tag{16}$$

Так как m является наибольшим целым числом, удовлетворяющим (16), оно представляет собой ближайшее целое, меньшее $2^p/d$, т.е.

$$\boxed{m = \frac{2^p - d - \text{rem}(2^p, d)}{d}}. \tag{17}$$

Объединяя это выражение с левой частью (16) и упрощая, получим

$$\boxed{2^p > n_c\left(d + \text{rem}(2^p, d)\right)}. \tag{18}$$

Доказательство пригодности предложенного в (17) и (18) алгоритма и корректности произведения выполняется аналогично доказательству для положительного делителя и здесь не приводится. Трудности возникают только при попытках доказать, что $-2^W \le m < 0$. Для доказательства рассмотрите по отдельности случаи, когда d представляет собой отрицательное число, абсолютное значение которого является степенью двойки. Для $d = -2^*$ можно легко показать, что $n_c = -2^{W-1} + 1$, $p = W + k - 1$ и $m = -2^{W-1} - 1$ (которое находится в изначально определяемом диапазоне). Для тех d, которые имеют отличный от -2^* вид, достаточно просто изменить доказательство, приведенное ранее, при рассмотрении положительного делителя.

Для каких делителей $m(-d) \ne -m(d)$?

Под $m(d)$ подразумевается множитель, соответствующий делителю d. Если $m(-d) = -m(d)$, код для деления на отрицательный делитель может быть сгенерирован путем вычисления множителя для $|d|$, изменения его знака и генерации кода, аналогичного коду из примера "деления на -7", проиллюстрированному ранее.

Сравнивая (18) с (6), а (17) с (5), можно увидеть, что если значение n_c для $-d$ представляет собой значение n_c для d, но с обратным знаком, то $m(-d) = -m(d)$. Следовательно, $m(-d) \ne -m(d)$ может быть только тогда, когда значение n_c, вычисленное для отрицательного делителя, представляет собой наибольшее по абсолютному значению отрицательное число -2^{W-1}. Такие делители представляют собой отрицательные сомножители $2^{W-1} + 1$. Эти числа встречаются очень редко, что иллюстрируется приведенными ниже разложениями.

$$2^{15}+1 = 3 \cdot 11 \cdot 331$$
$$2^{31}+1 = 3 \cdot 715827883$$
$$2^{64}+1 = 3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77158673929$$

Для *всех* этих множителей $m(-d) \ne -m(d)$. Вот набросок доказательства. Для $d > 0$ мы имеем $n_c = 2^{W-1} - d$. Поскольку $\text{rem}(2^{W-1}, d) = d - 1$, значение $p = W - l$ удовлетворяет (6), а следовательно, этому неравенству удовлетворяет и $p = W$. Однако для $d < 0$ мы имеем $n_c = -2^{W-1}$ и $\text{rem}(2^{W-1}, \text{rf}) = |d| - 1$. Следовательно, ни $p = W - 1$, ни $p = W$ не удовлетворяют (18), так что $p > W$.

10.6. Встраивание в компилятор

Для того чтобы компилятор мог заменить деление на константу произведением, он должен вычислить для данного делителя d магическое число M и величину сдвига s. Простейший способ состоит в вычислении (6) или (18) для $p = W, W+1, \ldots$ до тех пор, пока условие будет выполняться. Затем из (5) или (17) вычисляется значение m. Магическое число M представляет собой не что иное, как интерпретацию m как знакового целого числа, а $s = p - W$.

Описанная ниже схема обрабатывает положительные и отрицательные d с помощью небольшого дополнительного кода, и позволяет избежать арифметических выражений с двойными словами.

Вспомним, что n_c задается следующим образом:

$$n_c = \begin{cases} 2^{W-1} + \text{rem}\left(2^{W-1}, d\right) - 1, & d > 0, \\ -2^{W-1} + \text{rem}\left(2^{W-1} + 1, d\right), & d < 0. \end{cases}$$

Следовательно, $|n_c|$ можно вычислить так:

$$t = 2^{W-1} + \begin{cases} 0, & d > 0, \\ 1, & d < 0, \end{cases}$$

$$|n_c| = t - 1 - \text{rem}\left(t, |d|\right).$$

Остаток должен вычисляться с использованием беззнакового деления, в соответствии со значениями аргументов. Здесь используется запись $\text{rem}\left(t, |d|\right)$, а не эквивалентная ей $\text{rem}\left(t, d\right)$ именно для того, чтобы подчеркнуть, что программа должна работать с двумя положительными (и беззнаковыми) аргументами.

Исходя из (6) и (18), значение p может быть вычислено из соотношения

$$2^p > |n_c|\left(|d| - \text{rem}\left(2^p, |d|\right)\right), \tag{19}$$

после чего $|m|$ можно вычислить следующим образом (используя (5) и (17)):

$$|m| = \frac{2^p + |d| - \text{rem}\left(2^p, |d|\right)}{|d|}. \tag{20}$$

Непосредственное вычисление $\text{rem}\left(2^p, |d|\right)$ в (19) требует применения "длинного деления" (деление $2W$-битового делимого на W-битовый делитель и получение W-битовых частного и остатка), причем это длинное деление должно быть *беззнаковым*. Однако имеется путь решения (19), который позволяет избежать использования длинного деления и может быть легко реализован в обычных языках программирования высокого уровня с использованием только W-битовой арифметики. Тем не менее нам потребуется беззнаковое деление и беззнаковое сравнение.

Вычислить гет(2',|</|) можно инкрементно, инициализируя две переменные q и r значениями частного и остатка от деления 2^p на \d\ при $p = W - 1$, а затем обновляя значения q и r по мере роста p.

В процессе поиска при увеличении значения p на 1 значения q и r изменяются следующим образом (см. теорему D5 (первая формула)):

```
q = 2*q;
r = 2*r:
if (r >= abs(d))
```

```
{
    q = q + 1;
    r = r - abs(d);
}
```

Из левой половины неравенства (4) и правой половины (16) вместе с доказанными границами для *m* следует, что $q = \lfloor 2^p / |d| \rfloor < 2^W$, так что *q* можно представить *W*-битовым беззнаковым числом. Кроме того, $0 < r < |d|$, так что *r* можно представить *W*-битовым знаковым или беззнаковым числом. (Внимание: промежуточный результат $2r$ может превысить $2^{W-1} - 1$, поэтому *r* должно быть беззнаковым числом, и выполняющиеся выше сравнения также должны быть беззнаковыми.)

Далее вычисляется $8 = \backslash d \backslash - r$. Оба члена разности представимы в виде *W*-битовых беззнаковых целых чисел (так же, как и разность $1 \le 5 < |d|$), так что данное вычисление не представляет никаких трудностей.

Для того чтобы избежать длинного умножения в (19), перепишем его в виде

$$\frac{2"}{"|d|} > \delta .$$

Величина $2^p / |n_c|$ представима в виде *W*-битового беззнакового целого числа (аналогично (7), из (19) может быть показано, что $2^p \le 2|n_c| \cdot |d|$ и если $d = -2^{W-1}$, то $n_c = -2^{W-1} + 1$ и $p = 2W - 2$, так что $2^p / |n_c| = 2^{2W-2} / (2^{W-1} - 1) < 2^W$ при $W > 3$). Эти вычисления так же, как и в случае $\text{rem}(2^p, |d|)$, могут быть инкрементными (при увеличении *p).* Сравнение для случая $2^p / |n_c| \ge 2^{W-1}$ (что может произойти при больших *d)* должно быть беззнаковым.

Для вычисления *m* не требуется непосредственное вычисление (20), при котором может понадобиться длинное деление. Заметим, что

$$\frac{2^p + |d| - \text{rem}(2^p, |d|)}{|d|} - \left\lfloor \frac{2^p}{|d|} \right\rfloor + 1 = q + 1 .$$

Проверка завершения цикла $2^p / |n_c| > 8$ вычисляется достаточно сложно. Величина $2^p / |n_c|$ доступна только в виде частного $q\backslash$ и остатка r_1. Величина $2^p / |n_c|$ может быть (а может и не быть) целой (эта величина является целой только для $d = 2^{W-2} + 1$ и некоторых отрицательных значений *d).* Проверка $2^p / |n_c| \le S$ может быть закодирована следующим образом:

$$q_1 < \delta \mid (q_1 = \delta \,\&\, r_1 = 0) .$$

Полностью процедура для вычисления M и s для данного d показана в листинге 10.1, где представлена программа на языке С для $W = 32$. Здесь есть несколько мест, где может произойти переполнение, однако если его игнорировать, то полученный результат оказывается корректным.

Листинг 10.1. Вычисление магического числа для знакового деления

```
struct ms
{
    int M;      // Магическое число
    int s;      // Величина сдвига
};
```

```
struct ms magic(int d)
{   // d должно удовлетворять условию
    // 2 <= d <= 2**31-1 или -2**31 <= d <= -2
    int p;
    unsigned ad, anc, delta, q1, r1, q2, r2, t;
    const unsigned two31 = 0x80000000;   // 2**31
    struct ms mag;
    ad = abs(d);
    t = two31 + ((unsigned)d >> 31);
    anc = t - 1 - t%ad; // Абсолютное значение nc
    p = 31;              // Инициализация p
    q1 = two31/anc;      // Инициализация q1 = 2**p/|nc|
    r1 = two31 - q1*anc;// Инициализация r1 = rem(2**p,|nc|)
    q2 = two31/ad;       // Инициализация q2 = 2**p/|d|
    r2 = two31 - q2*ad; // Инициализация r2 = rem(2**p,|d|)
    do {
        p = p + 1;
        q1 = 2*q1;        // Обновление q1 = 2**p/|nc|.
        r1 = 2*r1;        // Обновление r1 = rem(2**p,|nc|)
        if (r1 >= anc)    // Здесь требуется беззнаковое
        {                 // сравнение
            q1 = q1 + 1;
            r1 = r1 - anc;
        }
        q2 = 2*q2;        // Обновление q2 = 2**p/|d|
        r2 = 2*r2;        // Обновление r2 = rem(2**p,|d|)
        if (r2 >= ad)     // Здесь требуется беззнаковое
        {                 // сравнение
            q2 = q2 + 1;
            r2 = r2 - ad;
        }
        delta = ad - r2;
    } while (q1 < delta || (q1 == delta && r1 == 0) ) ;
    mag.M = q2 + 1;
    if (d < 0)
        mag.M = -mag.M; // Магическое число и
    mag.s = p - 32;     // величина сдвига
    return mag;
}
```

Для использования результата этой программы компилятор должен сгенерировать команды `li` и `mulhs`, команду add при d > 0 и M < 0 или sub при d < 0 и M > 0 и, наконец, команду `shrsi` при s > 0. После этого генерируются команды `shri` и add.

В случае $W = 32$ можно избежать обработки отрицательного делителя, если использовать предвычисленный результат для $d=3$ и $d=715827883$, а для остальных отрицательных делителей использовать формулу $m(-d) = -m(d)$. Однако при этом программа из листинга 10.1 не станет значительно короче (если вообще сократится).

10.7. Дополнительные вопросы

Теорема DC2. *Наименьший множитель m нечетен, если p не равно W в принудительном порядке.*

Доказательство. Предположим, что уравнения (1,а) и (1,б) выполняются при наименьшем (не установленном принудительно) значении p и четном m. Очевидно, что тогда m можно разделить на 2, а p уменьшить на 1 и уравнения останутся справедливы. Но это противоречит предположению о том, что p — наименьшее значение, при котором выполняются данные уравнения.

Единственность

Магическое число для данного делителя может быть единственным (как, например, в случае *W=32,* rf=7), но зачастую это не так. Более того, эксперименты указывают, что как раз обычно имеется несколько магических чисел для одного делителя. Например, для $W=32, d=6$ имеется четыре магических числа:

$$M = 715827833 \qquad \left(\left(2^{32}+2\right)/6\right) \qquad s=0$$
$$M = 1431655766 \qquad \left(\left(2^{32}+2\right)/3\right) \qquad s=1$$
$$M = -1431655765 \qquad \left(\left(2^{33}+1\right)/3-2^{32}\right) \qquad s=2$$
$$M = -1431655764 \qquad \left(\left(2^{33}+4\right)/3-2^{32}\right) \qquad s=2.$$

Однако есть следующая теорема.

Теорема DC3. *Для данного делителя d существует только один множитель m, имеющий минимальное значение p, если p не приравнивается к W в принудительном порядке.*

Доказательство. Рассмотрим сначала случай $d>0$. Разность между верхней и нижней границами в неравенстве (4) составляет $2^p/dn_c$. Мы уже доказали (7), т.е. что если p минимально, то $2^p/dn_c<2$. Таким образом, может быть не более двух значений *m,* которые удовлетворяют неравенству (4). Пусть m равно минимальному из этих значений, задаваемому соотношением (5); тогда второе допустимое значение — $m+1$.

Пусть p_0 — наименьшее значение *p,* для которого от $+1$ удовлетворяет правой части неравенства (4) (значение p_0 не приравнено к W в принудительном порядке). Тогда

$$\frac{2^{p_0}+d-\mathrm{rem}\left(2^{p_0},d\right)}{d} < \frac{2''}{d}\frac{n_c+1}{n_c}.$$

Это выражение упрощается до

$$2^{p_0} > n_c\left(2d-\mathrm{rem}\left(2^{p_0},d\right)\right).$$

Деление на 2 дает

$$2^{p_0-1} > n_c\left(d-\frac{1}{2}\mathrm{rem}\left(2^{p_0},d\right)\right).$$

Поскольку в соответствии с теоремой D5 из раздела 9.1 $\mathrm{rem}(2^{ft},d) < 2\,\mathrm{rem}\left(2^{p_0-1},d\right)$, получаем

$$2^{p_0-1} > n_c\left(d-\mathrm{rem}\left(2^{p_0-1},d\right)\right),$$

что противоречит предположению о минимальности p_0.

Доказательство для случая $d<0$ аналогично и здесь не приводится.

Делители с лучшими программами

Программа для $d = 3$, $W = 32$ оказывается короче программы для общего случая, поскольку в ней не требуется выполнение команд add и shrsi после команды mulhs. Возникает вопрос: для каких делителей программа также оказывается укороченной?

Рассмотрим только положительные делители. Итак, требуется найти целые числа m и p, которые удовлетворяют уравнениям (1,а) и (1,б) и для которых выполняются соотношения $p = W$ и $0 < m \le 2^{W-1}$. Поскольку любые целые числа m к p, которые удовлетворяют уравнениям (1,а) и (1,б), должны также удовлетворять неравенству (4), достаточно найти те делители d, для которых (4) имеет решение при $p = W$ и $0 < m < 2^{W-1}$. Все решения (4) при $p = W$ задаются уравнением

$$m = \frac{2^W + kd - \text{rem}\left(2^W, d\right)}{d}, \quad k = 1, 2, 3\ldots$$

Объединяя его с правой частью (4) и упрощая, получим

$$\text{rem}\left(2^W, d\right) > kd - \frac{2^W}{n_c}. \qquad (21)$$

Наименьшим ограничением на $\text{rem}\left(2^W, d\right)$ является ограничение при $k = 1$ и n_c, равном своему минимальному значению 2^{W-2}, так что

$$\text{rem}\left(2^W, d\right) > d - 4,$$

т.е. d является делителем $2^W + 1$, $2^W + 2$ или $2^W + 3$.

Посмотрим теперь, какие из данных делителей в действительности имеют оптимальные программы.

Если d является делителем $2^W + 1$, то $\text{rem}\left(2^W, d\right) = d - 1$. Тогда решением (6) является $p = W$, так как неравенство превращается в очевидное:

$$2^W > n_c\left(d - (d - 1)\right) = n_c,$$

поскольку $n_c < 2^{W-1}$. При вычислении m имеем

$$m = \frac{2^W + d - (d - 1)}{d} = \frac{2^W + 1}{d},$$

и при $d > 3$ это значение оказывается меньше, чем 2^{W-1} (d не может быть равно 2, поскольку является делителем $2^W + 1$). Следовательно, все делители $2^W + 1$ имеют оптимальные программы.

Аналогично, если d является делителем $2^W + 2$, то $\text{rem}\left(2^W, d\right) = d - 2$. Здесь также решением (6) является $p - W$, поскольку неравенство также превращается в

$$2^W > n_c\left(d - (d - 2)\right) = 2n_c,$$

которое, очевидно, является истинным. Вычисление m дает значение

$$m = \frac{2^W + d - (d - 2)}{d} = \frac{2^W + 2}{d},$$

превышающее $2^{W-1}-1$ для d=2, но не превышающее $2^{W-1}-1$ при $W \ge 3$, $d \ge 3$ (случай W=3и d=3 невозможен, поскольку 3 не является делителем $2^3+2=10$). Следовательно, все делители 2^W+2, за исключением числа 2 и дополнительного ему (дополнительным к 2 делителем является $(2^W+2)/2$, т.е. число, которое нельзя представить как W-битовое знаковое целое).

Если d является делителем 2^W+3, то показать, что оптимальной программы для него нет, можно следующим образом. Поскольку $\text{rem}(2^W,d)$-d -3 , из неравенства (21) имеем

$$n_c < \frac{2^W}{kd-d+3},\ k=1,2,3,\ldots$$

Самое слабое ограничение — при k= 1, так что $n_c < 2^W/3$.

Из (3,а) $n_c \ge 2^{W-1}$ - d или $d \ge 2^{W-1}$ - n_c, следовательно:

$$d > 2^{W-1} -- \frac{2^W}{3} = \frac{2^W}{6}.$$

Кроме того, поскольку 2, 3 и 4 не могут быть делителями 2^W+3, наименьшим возможным делителем 2^W+3 является 5 и, следовательно, наибольший возможный делитель равен $(2^W+3)/5$. Таким образом, если d является делителем 2^W+3 и d имеет оптимальную программу, то

$$\frac{2^W}{6} < d \le \frac{2^W+3}{5}.$$

Преобразуя данное неравенство, для дополнительного к d делителя $(2^W+3)/d$ получаем следующие границы:

$$5 \le \frac{2^W+3}{d} < \frac{(2^W+3)\cdot 6}{2^W} = 6 + \frac{18}{2^W}.$$

Эти границы указывают, что при $W \ge 5$ единственными возможными дополнительными делителями являются 5 и 6. При $W < 5$ легко убедиться, что делителей 2^W+3 не существует. Поскольку 6 не может быть делителем 2^W+3, единственным возможным делителем этого числа может быть 5. Таким образом, единственным возможным делителем 2^W+3, который может иметь оптимальную программу, является $(2^W+3)/5$.

Для $d=(2^W+3)/5$

$$n_c = \left\lfloor \frac{2^{W-1}}{(2^W+3)/5} \right\rfloor \left(\frac{2^W+3}{5}\right) - 1,$$

а так как для $W \ge 4$

$$2 < \frac{2^{W-1}}{(2^W+3)/5} < 2.5,$$

то

$$n_c = 2\left(\frac{2^W+3}{5}\right) - 1.$$

Эта величина превышает $2^W/3$, так что $d = (2^W+3)/5$ оптимальной программы не имеет. Поскольку для $W < 4$ величина $2^W + 3$ делителей не имеет, можно сделать вывод о том, что делителей $2^W + 3$ с оптимальной программой не существует.

Резюмируем: все делители $2^W + 1$ и $2^W + 2$, за исключением 2 и $(2^W + 2)/2$, имеют оптимальные программы, и никакие другие числа оптимальных программ не имеют. Более того, приведенное выше доказательство показывает, что алгоритм из листинга 10.1 всегда дает оптимальную программу, если таковая существует.

Рассмотрим частные случаи значений W, равные 16, 32 и 64. Соответствующие разложения на множители показаны ниже.

$2^{16} + 1 =$ 65537 (простое)
$2^{16} + 2 =$ $2 \cdot 3^2 \cdot 11 \cdot 331$
$2^{32} + 1 =$ $641 \cdot 6700417$
$2^{32} + 2 =$ $2 \cdot 3 \cdot 715827883$
$2^{64} + 1 =$ $274177 \cdot 67280421310721$
$2^{w} + 2 =$ $2 \cdot 3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77158673929$

Следовательно, для $W = 16$ имеется 20 делителей, для которых существует оптимальная программа. Из них меньше 100 делители 3, 6, 9, 11, 18, 22, 33, 66 и 99.

Для $W = 32$ есть шесть таких делителей: 3, 6, 641, 6700417, 715827883, 1431655766.

Для $W = 64$ есть 126 таких делителей. Из них меньше 100 делители 3, 6, 9, 18, 19, 27, 38, 43, 54, 57 и 86.

10.8. Беззнаковое деление

Беззнаковое деление на величину, представляющую собой степень двойки, реализуется с помощью единственной команды *сдвига вправо,* а остаток — с помощью команды *и с непосредственно задаваемым операндом.*

Может показаться, что обработка других делителей должна быть несложной: просто использовать результаты для знакового деления с $d > 0$, опуская две команды, которые добавляют 1 при отрицательном частном. Однако вы увидите, что некоторые детали беззнакового деления в действительности несколько сложнее.

Беззнаковое деление на 3

Начнем рассмотрение беззнакового деления на другие числа с беззнакового деления на 3 на 32-битовой машине. Поскольку делимое n теперь может достигать по величине $2^{32} - 1$, множитель $(2^{32} + 2)/3$ оказывается неадекватен (член-ошибка $2n/(3 \cdot 2^{32})$ может превысить $1/3$). Однако можно использовать множитель $(2^{33} + 1)/3$. Соответствующий код имеет вид

```
li     M,0xAAAAAAAB  Загрузка магического числа (2**33+1)/3
mulhu q,M,n          q = floor(M*n/2**32)
shri q,q/1

muli  t,q,3          Вычисление остатка по формуле
sub   r,n,t          r = n - q*3
```

В этом случае нам требуется команда mulhu, которая возвращает старшие 32 бита беззнакового 64-битового произведения.

Чтобы убедиться в корректности приведенного кода, заметим, что он вычисляет следующее значение:

$$q = \left\lfloor \frac{2^{33}+1}{3} \frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{n}{3 \cdot 2^{33}} \right\rfloor.$$

При $0 \le n < 2^{32}$ выполняется неравенство $0 \le n/(3 \cdot 2^{33}) < 1/3$, так что в соответствии с теоремой D4 $q = \lfloor n/3 \rfloor$.

При вычислении остатка может произойти переполнение при выполнении команды *умножения на непосредственно заданное значение,* если операнды рассматриваются как знаковые целые числа, но если считать операнды и результат беззнаковыми, то переполнение при выполнении данной команды невозможно. Переполнение невозможно и при выполнении команды *вычитания,* так как результат находится в диапазоне от 0 до 2, поэтому получаемый таким образом остаток корректен.

Беззнаковое деление на 7

При беззнаковом делении на 7 на 32-битовом компьютере множители $(2^{32}+3)/7$, $(2^{33}+6)/7$ и $(2^{34}+5)/7$ оказываются неадекватными, поскольку дают слишком большой член-ошибку. Можно использовать множитель $(2^{35}+3)/7$, но он слишком велик для представления 32-битовым беззнаковым числом. Умножение на это число можно выполнить посредством умножения на $(2^{35}+3)/7 - 2^{32}$ с последующей коррекцией путем сложения. Соответствующий код имеет следующий вид:

```
li    M,0x24924925  Магическое число (2**35+3)/7 - 2**32
mulhu q,M,n          q = floor(M*n/2**32)
add   q,q,n         Может вызвать переполнение (перенос)
shrxi q,q,3         Сдвиг вправо с битом переноса

muli  t,q,7          Вычисление остатка по формуле
sub   r,n,t          r = n - q*7
```

Здесь возникает небольшая проблема: команда add может вызвать переполнение. Для того чтобы обойти эту ситуацию, была придумана новая команда *расширенного сдвига вправо на непосредственно заданную величину* (shrxi), которая рассматривает бит переноса команды add и 32 бита регистра q как единую 33-битовую величину, и выполняет ее сдвиг вправо с заполнением освободившихся разрядов нулевыми битами. В семействе процессоров Motorola 68000 эти действия можно реализовать при помощи двух команд: *расширенного циклического сдвига* вправо на один бит с последующим *беззнаковым сдвигом вправо* на 3 бита (команда rorx на самом деле использует **X-бит,** но команда add присваивает ему то же значение, что и биту переноса). На большинстве компьютеров для выполнения этих действий требуется большее количество команд. Например, на PowerPC требуется выполнение трех команд: сброс правых трех битов q, прибавление бита переноса к q и циклический сдвиг вправо на три позиции.

При той или иной реализации команды shrxi приведенный выше код вычисляет

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{35}+3}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) \Big/ 2^3 \right\rfloor = \left\lfloor \frac{n}{3} + \frac{3n}{7 \cdot 2^{35}} \right\rfloor.$$

При $0 < n < 2^{32}$ выполняется неравенство $0 < 3n/(7 \cdot 2^{35}) < 1/7$, так что в соответствии с теоремой *D4* $q = \lfloor n/7 \rfloor$.

Гранлунд (Granlund) и Монтгомери (Montgomery) [21] предложили остроумную схему, позволяющую избежать применения команды shrxi. Она требует того же количества команд, что и рассмотренная выше схема, но использует только элементарные команды, которые есть практически у каждого компьютера; переполнения при этом оказываются просто невозможны. Схема использует следующее тождество:

$$\left\lfloor \frac{q+n}{2^p} \right\rfloor = \left\lfloor \left(\left\lfloor \frac{n-q}{2} \right\rfloor + q \right) \Big/ 2^{p-1} \right\rfloor, \quad p \geq 1.$$

Применим данное тождество к нашей задаче, полагая $q = \lfloor Mn/2^{32} \rfloor$, где $0 \leq M < 2^{32}$. Вычитание не может вызвать переполнения, поскольку

$$0 \leq q = \left\lfloor \frac{Mn}{2^{32}} \right\rfloor \leq n,$$

поэтому очевидно, что $0 < n - q < 2^{32}$. Сложение также не может вызвать переполнения, поскольку

$$\left\lfloor \frac{n-q}{2} \right\rfloor + q = \left\lfloor \frac{n-q}{2} + q \right\rfloor = \left\lfloor \frac{n+q}{2} \right\rfloor$$

и $0 \leq n, q < 2^{32}$.

Использование предложенных идей приводит к следующему коду для беззнакового деления на 7:

```
li    M,0x24924925  Магическое число  (2**35+3)/7 - 2**32
mulhu q,M,n         q = floor(M*n/2**32)
sub   t,n,q         t = n - q
shri  t,t,1         t = (n - q)/2
add   t,t,q         t = (n - q)/2 + q = (n + q)/2
shri  q,t,2         q = (n+Mn/2**32)/8 = floor(n/7)

muli  t,q,7         Вычисление остатка по формуле
sub   r,n,t         r = n - q*7
```

Чтобы эта схема работала, величина сдвига в гипотетической команде shrxi должна быть больше 0. Можно показать, что если $d > 1$ и множитель $m > 2^{32}$ (т.е. необходима команда shrxi), то величина сдвига больше 0.

10.9. Беззнаковое деление на делитель, не меньший 1

Для данного размера слова $W \geq 1$ и делителя $1 < d < 2^W$ требуется найти наименьшее целое m и целое p, такие, что

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \text{ для } 0 \leq n < 2^W, \qquad (22)$$

при этом $0 \le m < 2^{W+1}$ и $p \ge W$.

В случае беззнакового деления магическое число M определяется как

$$M = \begin{cases} m, & 0 \le m < 2^W, \\ m - 2^W, & 2^W \le m < 2^{W+1}. \end{cases}$$

Поскольку (22) должно выполняться для $n = d$, $\lfloor md/2^p \rfloor = 1$ или

$$\frac{md}{2^p} \quad . \tag{23}$$

Пусть, как и в случае знакового деления, n_c представляет собой наибольшее значение n, такое, что $\text{rem}(n_c, d) = d - 1$. Его можно вычислить как $n_c = \lfloor 2^W/d \rfloor d - 1 = 2^W - \text{rem}(2^W, d) - 1$. Тогда

$$2^W - d \le n_c \le 2^W - 1 \tag{24,а}$$

и

$$n_c \ge d - 1. \tag{24,б}$$

Это означает, что $n_c > 2^{W-1}$.

Поскольку (22) должно выполняться при $n = n_c$,

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d}$$

или

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}.$$

Объединение этого неравенства с (23) дает

$$\frac{2^p}{d} \le m < \frac{2^p}{d} \frac{n_c + 1}{n_c}. \tag{25}$$

Поскольку m представляет собой наименьшее целое, удовлетворяющее неравенству (25), оно должно быть ближайшим целым числом, большим или равным $2^p/d$, т.е.

$$\boxed{m = \frac{2^p + d - 1 - \text{rem}(2^p - 1, d)}{d}}. \tag{26}$$

Комбинируя это уравнение с правой частью (25) и упрощая, получим

$$\boxed{2^p > n_c\left(d - 1 - \text{rem}(2^p - 1, d)\right)}. \tag{27}$$

Беззнаковый алгоритм

Таким образом, алгоритм состоит в поиске методом проб и ошибок минимального $p > W$, удовлетворяющего неравенству (27), после чего из уравнения (26) вычисляется m. Это наименьшее возможное значение m, удовлетворяющее (22) при $p \ge W$. Как и в знаковом случае, если неравенство (27) выполняется для некоторого значения p, то оно выполняется и для всех больших значений p. Доказательство этого факта, по сути, ничем не

отличается от доказательства теоремы DC1, только вместо первой формулы из теоремы D5 в доказательстве используется вторая.

Доказательство пригодности беззнакового алгоритма

Покажем, что (27) всегда имеет решение и что $0 \le m < 2^{W+1}$.

Поскольку для любого неотрицательного числа x имеется степень 2, большая x, но не превышающая $2x+1$, из (27) получаем

$$n_c\left(d-1-\text{rem}\left(2^p-1,d\right)\right) < 2^p \le 2n_c\left(d-1-\text{rem}\left(2^p-1,d\right)\right)+1 .$$

Так как $0 \le \text{rem}\left(2^{p-1},d\right) \le d-1$,

$$1 \le 2^p \le 2n_c\left(d-1\right)+1 . \qquad (28)$$

С учетом того, что $n_c, d \le 2^W-1$, это неравенство превращается в

$$1 \le 2^p \le 2\left(2^W-1\right)\left(2^W-2\right)+1$$

или

$$0 \le p \le 2W . \qquad (29)$$

Таким образом, неравенство (27) всегда имеет решение.

Если p не приравнивается принудительно к W, то из (25) и (28) следует

$$\frac{1}{d} \le m < \frac{2n_c\left(d-1\right)+1}{d}\frac{n_c+1}{n_c},$$

$$1 \le m < \frac{2d-2+1/n_c}{d}\left(n_c+1\right),$$

$$1 \le m < 2\left(n_c+1\right) \le 2^{W+1} .$$

Если же p в принудительном порядке приравнено к W, то из (25) вытекает

$$\frac{2^W}{d} \le m < \frac{2^W}{d}\frac{n_c+1}{n_c} .$$

В силу того, что $1 \le d \le 2^W-1$ и $n_c \ge 2^{W-1}$,

$$\frac{2^W}{2^W-1} \le m < \frac{2^W}{1}\frac{2^{W-1}+1}{2^{W-1}},$$

$$2 \le m \le 2^W+1 .$$

Следовательно, в любом случае от находится в пределах схемы, проиллюстрированной примером беззнакового деления на 7.

Доказательство корректности произведения в беззнаковом случае

Покажем, что если p и m вычислены на основании формул (27) и (26), то выполняется уравнение (22).

Легко показать, что уравнение (26) и неравенство (27) приводят к неравенству (25), которое практически тождественно неравенству (4); остальная часть доказательства, по сути, идентична доказательству для знакового деления при $n > 0$.

10.10. Встраивание в компилятор при беззнаковом делении

В реализации алгоритма, основанного на непосредственных вычислениях, использованных в доказательстве, имеются трудности. Хотя, как доказано выше, $p < 2W$, случай $p = 2W$ не исключается (например, для $d = 2^W - 2$, $W \geq 4$). Если $p = 2W$, вычислить m становится сложно в силу того, что делимое в (26) не размещается в $2W$-битовом слове.

Однако реализация этих действий возможна при использовании методики "инкрементного деления и взятия остатка", уже рассматривавшейся нами ранее и примененной для данного случая в алгоритме, который приведен в листинге 10.2 (для случая $W=32$). В этом алгоритме, помимо магического числа и величины сдвига, возвращается индикатор необходимости генерации команды add (в случае знакового деления необходимость добавления этой команды распознавалась по тому, что M и d имели разные знаки).

Листинг 10.2. Вычисление магического числа для случая беззнакового деления

```
struct mu
{
    unsigned M;   // Магическое число
    int a;        // Индикатор "add"
    int s;        // Величина сдвига
};

struct mu magicu(unsigned d)
{
    // d должно быть в границах 1 <= d <= 2**32-1
    int p;
    unsigned nc, delta, q1, r1, q2, r2;
    struct mu magu;
    magu.a = 0;          // Инициализация индикатора "add"
    nc = -1 - (-d)%d; // Используется беззнаковая арифметика
    p = 31;                    // Инициализация p.
    q1 = 0x80000000/nc;       // Инициализация q1=2**p/nc
    r1 = 0x80000000 - q1*nc;// Инициализация r1=rem(2**p,nc)
    q2 = 0x7FFFFFFF/d;        // Инициализация q2=(2**p-1)/d
    r2 = 0x7FFFFFFF - q2*d; // Инициализация r2=rem(2**p-1,d)
    do {
        p = p + 1;
        if (r1 >= nc - r1)
        {
            q1 = 2*q1 + 1;    // Коррекция q1
            r1 = 2*r1 - nc;   // Коррекция r1
        }
        else
        {
            q1 = 2*q1;
            r1 = 2*r1;
        }
        if (r2 + 1 >= d - r2)
        {
            if (q2 >= 0x7FFFFFFF) magu.a = 1;
            q2 = 2*q2 + 1;      // Коррекция q2
            r2 = 2*r2 + 1 - d; // Коррекция r2
        }
        else
```

```
      {
          if (q2 >= 0x80000000) magu.a = 1;
          q2 = 2*q2;
          r2 = 2*r2 + 1;
      }
      delta = d - 1 - r2 ;
   } while (p < 64 &&
           (q1< delta | |
            (q1 == delta && r1 == 0)));
   magu.M = q2 + 1;  // Возвращаемые магическое число
   magu.s = p - 32;  // и величина сдвига
   return magu;      //  (magu.a установлено ранее)
}
```

Приведем ряд ключевых моментов в понимании этого алгоритма.

- В ряде мест алгоритма может произойти беззнаковое переполнение, которое должно быть проигнорировано.
- $n_c = 2^W - \text{rem}(2^W, d) - 1 = (2^W - 1) - \text{rem}(2^W - d, d)$.
- Частное и остаток от деления 2^p на n_c не могут быть подкорректированы так же, как и в алгоритме из листинга 10.1, поскольку здесь величина `2*r1` может вызвать переполнение. Следовательно, алгоритм должен выполнить проверку "if `(r1>=nc-r1)`" вместо более естественной проверки "if `(2*r1>=nc)`". Такое же замечание применимо и к вычислению частного и остатка от деления $2^p - 1$ на d.
- $0 < \delta < d - 1$, так что δ представимо в виде 32-битового беззнакового целого числа.
- $m = \left(2^p + d - 1 - \text{rem}(2^p - 1, d)\right)/d = \left\lfloor (2^p - 1)/d \right\rfloor + 1 = q_2 + 1$.
- В программе не выполняется явное вычитание 2^W для случая, когда магическое число M превышает $2^W - 1$; это вычитание происходит в случае, когда вычисление q2 вызывает переполнение.
- Индикатор `magu.a` необходимости команды add из-за переполнения не может быть установлен путем непосредственного сравнения M с 2^{32} или q2 с $2^{32} - 1$. Вместо этого программа проверяет q2 до того, как может произойти переполнение. Если q2 достигнет величины $2^{32} - 1$, так что M станет не менее 2^{32}, то индикатору `magu.a` будет присвоено значение 1; в противном случае это значение останется равным 0.
- Неравенство (27) эквивалентно неравенству $2^p/n_c > \delta$.
- Проверка `p < 64` в условии цикла необходима постольку, поскольку переполнение ql может привести к тому, что будет выполнено слишком большое количество итераций и будет получен неверный результат.

10.11. Дополнительные вопросы (беззнаковое деление)

Теорема DC2U. *Наименьший множитель m нечетен, если p принудительно не присваивается значение W.*

Теорема DC3U. *Для данного делителя d существует только один множитель m, имеющий минимальное значение p, если p не приравнивается к W в принудительном порядке.*

Доказательства этих теорем практически идентичны доказательствам соответствующих теорем для знакового деления.

Делители с лучшими программами (беззнаковое деление)

Для того чтобы найти в случае беззнакового деления делители (если таковые имеются) с оптимальными программами поиска частного, состоящими из двух команд (`li`, `mulhu`), можно выполнить почти такой же анализ, как и для случая знакового деления (см. выше раздел "Делители с лучшими программами"). В результате анализа выясняется, что такими делителями являются делители чисел 2^W и 2^W+1 (за исключением $d=1$). Для распространенных размеров слов таких нетривиальных делителей оказывается очень мало. При использовании 16-битовых слов таких делителей нет вовсе; для 32-битовых слов их всего два— 641 и 6700417. Два таких делителя (274177 и 67280421310721) есть и при работе с 64-битовыми словами.

Случай $d=2^k, * = 1,2,\ldots$ заслуживает отдельного упоминания. В этом случае рассмотренный алгоритм дает нам $p=W$ (принудительное присвоение) и $m=2^{32-k}$. Это минимальное значение *m,* но не минимальное значение *M*. Наилучший код получается при использовании $p=W+k$. Тогда $m=2^W$, *M* равно 0, a — единице, *s* — *k*. Сгенерированный код включает умножение на 0 и может быть упрощен до одной команды *сдвига вправо на величину k*. На практике делители, представляющие собой степень 2, стоит рассматривать как особый случай и не привлекать для их обработки программу `magicu`. (Этого не происходит при знаковом делении, поскольку в этом случае *m* не может быть степенью 2. **Доказательство.** При $d>0$ неравенство (4), будучи скомбинированным с неравенством (3,б), дает $d-1<2^p/m<d$.Следовательно, $2^p/m$ не может быть целым числом. При $d<0$ аналогичный результат получается при комбинировании неравенств (16) и (15,б).)

При беззнаковом делении, когда компьютер не имеет команды `shrxi`, код для случая $m>2^W$ оказывается значительно хуже, чем код для $m<2^W$. Поэтому возникает вопрос, как часто приходится иметь дело с большими множителями. Для размера слова 32 бита среди целых чисел, меньших 100, имеется 31 "плохой" делитель: 1, 7, 14, 19, 21, 27, 28, 31, 35, 37, 38, 39, 42, 45, 53, 54, 55, 56, 57, 62, 63, 70, 73, 74, 76, 78, 84, 90, 91, 95 и 97.

Использование знакового деления вместо беззнакового и наоборот

Если ваш компьютер не оснащен командой mulhu, но имеет команду `mulhs` (или команду знакового длинного умножения), то можно воспользоваться приемом, о котором говорилось в разделе 8.3.

В разделе 8.3 была приведена последовательность из семи команд, которая позволяет получить результат выполнения команды mulhu из команды mulhs. Однако в нашей ситуации этот прием упрощается, поскольку магическое число *M* известно заранее, так что компилятор может протестировать старший бит заранее и сгенерировать в соответствии с его значением ту или иную последовательность действий после выполнения команды "mulhu q, M, n". Здесь t обозначает временный регистр.

$M_{31} = 0$	$M_{31} = 1$
mulhs q,M,n shrsi t,n,31 and t,t,M add q,q,t	mulhs q,M,n shrsi t,n,31 and t,t,M add t,t,n add q,q,t

С учетом других команд, используемых вместе с mulhu, всего для получения частного и остатка от беззнакового деления на константу на компьютере, не оснащенном беззнаковым умножением, требуется от шести до восьми команд.

Этот прием можно обратить для получения команды mulhs из команды mulhu. При этом получается практически такой же код, только команда mulhs заменяется командой mulhu, а последние команды add в обоих столбцах заменяются командами sub.

Более простой беззнаковый алгоритм

Отказ от требования минимальности магического числа приводит к более простому алгоритму. Вместо (27) можно использовать

$$2^p \geq 2^W \left(d - 1 - \text{rem}\left(2^p - 1, d\right)\right), \tag{30}$$

а затем вычислить m, как и ранее, по формуле (26).

Должно быть очевидно, что формально этот алгоритм корректен (т.е. вычисленное значение m удовлетворяет уравнению (22)), поскольку единственное его отличие от предыдущего алгоритма состоит в том, что он вычисляет значение p, которое для некоторых значений d не обязательно будет большим. Можно доказать, что значение m, вычисленное по (30) и (26), меньше 2^{W+1}. Здесь доказательство опускается и просто приводится конечный результат — алгоритм, представленный в листинге 10.3.

Листинг 10.3. Упрощенный алгоритм вычисления магического числа для беззнакового деления

```
struct mu
{
    unsigned M;  // Магическое число
    int a;       // Индикатор "add"
    int s;       // Величина сдвига
};

struct mu magicu2(unsigned d)
{
    // d должно быть в границах  1 <= d <= 2**32-1
    int p;
    unsigned p32, q, r, delta;
    struct mu magu;
    magu.a = 0;           // Инициализация индикатора "add"
    p = 31;                // Инициализация p
    q = 0x7FFFFFFF/d;      // Инициализация q =  (2**p-1)/d
    r = OX7FFFFFFF - q*d; // Инициализация r = rem(2**p-1,d)
    do {
        p = p + 1;
        if (p == 32) p32 = 1; // p32 = 2**(p-32).
        else p32 = 2*p32;
        if (r + 1 >= d - r)
```

```
        {
            if (q >= OX7FFFFFFF) magu.a = 1;
            q = 2*q + 1;      // Коррекция q
            r = 2*r + 1 - d; // Коррекция r
        }
        else
        {
            if (q >= 0x80000000) magu.a = 1;
            q = 2*q;
            r = 2*r + 1;
        }
        delta = d - 1 - r;
    } while (p < 64 && p32 < delta);
    magu.M = q + 1;  // Возвращаемые магическое число
    magu.s = p - 32; // и величина сдвига
    return magu;     // (magu.a установлено ранее)
}
```

Алверсон [2] предложил более простой алгоритм, который рассматривается в следующем разделе, но он иногда дает очень большие значения m. Главное в алгоритме `magicu2` в том, что он почти всегда дает минимальное значение m при $d \le 2^{W-1}$. Если размер слова равен 32 битам, наименьший делитель, для которого `magicu2` не дает минимального множителя, равен 102807 (в этом случае magicu дает значение m, равное 2737896999, а `magicu2` — значение 5475793997).

Существует аналог алгоритма `magicu2` и для знакового деления на положительные делители, но он плохо работает для знакового деления на произвольные делители.

10.12. Применение к модульному делению и делению с округлением к меньшему значению

Может показаться, что преобразовать в умножение модульное деление на константу или деление с округлением к меньшему значению — задача еще более простая, чем при делении с отсечением, и для этого достаточно всего лишь убрать шаг "добавить 1, если делимое отрицательно". Однако это не так. Предложенные выше методы не применимы к другим типам деления путем простых и очевидных преобразований. Вероятно, разработанное преобразование такого типа будет включать изменение множителя m в зависимости от знака делимого.

10.13. Другие похожие методы

Вместо кодирования алгоритма magic можно воспользоваться таблицей с магическими числами и величинами сдвигов для некоторых небольших делителей. Делители, равные табличным значениям, умноженным на степень 2, легко обрабатываются следующим образом.

1) Подсчитывается количество завершающих нулевых битов в d, которое далее обозначается как k.
2) В качестве аргумента поиска в таблице используется значение $d/2^k$ (сдвиг вправо на k позиций).
3) Используем магическое число, найденное в таблице.
4) Используем найденную в таблице величину сдвига, увеличенную на k.

Таким образом, если в таблице имеются делители 3, 5, 25, то могут быть обработаны делители 6, 10, 100 и т.д.

Такая процедура обычно дает наименьшее магическое число, хотя и не всегда. Если размер слова равен 32 бита, то наименьший положительный делитель, когда такое происходит, равен 334972. Данный метод при этом дает значения $m=3361176179$ и $s=18$; минимальное же магическое число для данного делителя составляет 840294045, а величина сдвига равна 16. Процедура также дает неминимальный результат при $d=-6$. В обоих приведенных случаях снижается качество генерируемого кода деления.

Алверсон [2] был первым, кто указал на корректность работы описываемого далее метода для всех делителей. Используя наши обозначения, можно сказать, что его метод для беззнакового целого деления на d состоит в использовании величины сдвига $p=W+\lceil \log_2 d \rceil$ и множителя $m=\lceil 2^p/d \rceil$ с последующим выполнением деления $n \div d = \lfloor mn/2^p \rfloor$ (т.е. путем *умножения* и *сдвига вправо)*. Он доказал, что множитель m меньше 2^{W+1} и что этот метод дает точное значение частного для всех n, выражаемых с помощью W-битового числа.

Метод Алверсона представляет собой более простой вариант нашего, в котором для определения p не используется поиск методом проб и ошибок, а следовательно, он в большей степени подходит для аппаратной реализации, в чем и состоит его главное назначение. Однако его множитель m всегда оказывается не меньшим, чем 2^W, а потому при программной реализации всегда требуется код, проиллюстрированный в примере деления на 7 (т.е. всегда требуются либо команды add и shrxi, либо четыре альтернативные команды). Поскольку большинство небольших делителей могут быть обработаны при помощи множителей, меньших 2^W, представляется разумным рассмотреть эти случаи.

В случае знакового деления Алверсон предложил искать множитель для $|d|$ и длины слова $W-1$ (тогда $2^{W-1} \le m < 2^W$), умножать на него делимое и, если операнды имеют противоположные знаки, изменять знак полученного результата. (Множитель должен быть таким, чтобы давать корректный результат для делимого 2^{W-1}, которое представляет собой абсолютное значение максимального отрицательного числа). Похоже, что такое предложение может дать код, лучший по сравнению с кодом для множителя $m>2^W$. Применяя данный прием к знаковому делению на 7, получим следующий код (где для того, чтобы избежать ветвления, использовано соотношение $-x=\bar{x}+1$):

```
abs    an,n
li     M,0x92492493   Магическое число (2**34+5)/7
mulhu  q,M,an         q = floor(M*an/2**32)
shri   q,q,2
shrsi  t,n,31         Эти три команды
xor    q,q,t          изменяют знак q,
sub    q,q,t          если n отрицательно
```

Этот код не столь хорош, как тот, который был предложен для знакового деления на 7 ранее (7 и 6 команд соответственно), но он может оказаться полезным на компьютере, у которого есть команды abs и mulhu и нет команды mulhs.

10.14. Некоторые магические числа

Таблица 10.1. Некоторые магические числа для $W = 32$

	Знаковое		Беззнаковое		
d	M (шестнадцатеричное)	s	M (шестнадцатеричное)	a	s
-5	99999999	1			
-3	55555555	1			
-2^k	7FFFFFFF	k-1			
1	-	-	0	1	0
2^k	80000001	k-1	2^{32-k}	0	0
3	55555556	0	AAAAAAAB	0	1
5	66666667	1	CCCCCCCD	0	2
6	2AAAAAAB	0	AAAAAAAB	0	2
7	92492493	2	24924925	1	3
9	38E38E39	1	38E38E39	0	1
10	66666667	2	CCCCCCCD	0	3
11	2E8BA2E9	1	BA2E8BA3	0	3
12	2AAAAAAB	1	AAAAAAAB	0	3
25	51EB851F	3	51EB851F	0	3
125	10624DD3	3	10624DD3	0	3
625	68DB8BAD	8	D1B71759	0	9

Таблица 10.2. Некоторые магические числа для $W = 64$

	Знаковое		Беззнаковое		
d	M (шестнадцатеричное)	s	M (шестнадцатеричное)	a	s
-5	99999999 99999999	1			
-3	55555555 55555555	1			
-2^k	7FFFFFFF FFFFFFFF	k-1			
1	-	-	0	1	0
2^k	80000000 00000001	k-1	2^{64-k}	0	0
3	55555555 55555556	0	AAAAAAAA AAAAAAAB	0	1
5	66666666 66666667	1	CCCCCCCC CCCCCCCD	0	2
6	2AAAAAAA AAAAAAAB	0	AAAAAAAA AAAAAAAB	0	2
7	49249249 24924925	1	24924924 92492493	1	3
9	1C71C71C 71C71C72	0	E38E38E3 8E38E38F	0	3
10	66666666 66666667	2	CCCCCCCC CCCCCCCD	0	3
11	2E8BA2E8 BA2E8BA3	1	2E8BA2E8 BA2E8BA3	0	1
12	2AAAAAAA AAAAAAAB	1	AAAAAAAA AAAAAAAB	0	3
25	A3D70A3D 70A3D70B	4	47AE147A E147AE15	1	5
125	20C49BA5 E353F7CF	4	0624DD2F 1A9FBE77	1	7
625	346DC5D6 3886594B	7	346DC5D6 3886594B	0	7

10.15. Точное деление на константу

Под "точным делением" подразумевается деление, о котором заранее известно, что его остаток равен 0. Хотя такая ситуация и не очень распространена, она имеет место, например при вычитании двух указателей в языке С. В нем результат вычитания $p - q$, где p и q — указатели, определен и переносим, только если p и q указывают на объекты в

одном и том же массиве [32, раздел 7.6.2]. Если размер элемента массива равен *s,* то код для вычисления разности двух указателей реально вычисляет $(p - q)/s$.

Материал этого раздела основан на [21, раздел 9].

Рассматриваемый здесь метод применим как к знаковому, так и к беззнаковому точному делению и базируется на следующей теореме.

Теорема MI. *Если a и m — взаимно простые целые числа, то существует целое число* $1 \le \bar{a} \le m$, *такое, что* $a\bar{a} \equiv 1 (\mathrm{mod}\ m)$.

Таким образом, $\bar{a}$ представляет собой обратный мультипликативный по модулю *m* элемент по отношению к *a.* Есть несколько способов доказательства этой теоремы, три из них описаны в [49]. Доказательство, приведенное далее, требует только знания основ теории сравнений.

Доказательство. Докажем более общее по сравнению с данной теоремой утверждение. Если *a* и *m* взаимно просты (и, следовательно, имеют ненулевые значения), то для множества *x,* состоящего из *m* различных по модулю *m* значений, значения *ax* также будут различны по модулю *m.* Например, при $a = 3$ и $m = 8$ и значениях *x* из диапазона от O до 7 получим, что значения *ax* представляют собой множество O, 3, 6, 9, 12, 15, 18, 21, или по модулю 8 это числа 0, 3, 6, 1, 4, 7, 2, 5. Заметим, что в полученной последовательности вновь представлены все числа от 0 до 7.

Чтобы доказать это в общем случае, воспользуемся методом доказательства от противного. Предположим, что существуют два различных по модулю *m* целых числа, которые отображаются в одно и то же число по модулю *m* при умножении на *a,* т.е. существуют такие *x* и *y* ($x \not\equiv y (\mathrm{mod}\ m)$), что $ax = ay (\mathrm{mod}\ m)$. Но в таком случае существует целое число *k*, такое, что

$$ax - ay = km, \text{ или}$$

$$a(x - y) = km.$$

Поскольку *a* не имеет общих множителей с *m,* разность $x - y$ должна быть кратной *m,* т.е.

$$x \equiv y (\mathrm{mod}\ m).$$

Однако этот вывод противоречит исходному предположению.

Теперь, поскольку значения *ax* принимают все *m* возможных различных значений по модулю *m*, среди значений *x* найдется такое, для которого *ax* по модулю *m* равно 1.

Приведенное доказательство показывает также, что имеется только одно значение *x* (по модулю *m),* такое, что $ax \equiv 1 (\mathrm{mod}\ m)$, т.е. что мультипликативное обращение однозначно. Можно также показать, что имеется единственное (по модулю т) целое число *x,* такое, что $ax = b (\mathrm{mod} m)$, где *b* — любое целое число.

В качестве примера рассмотрим случай $m = 16$. Тогда $\bar{3} = 11$, так как $3 \cdot 11 = 33 = 1 (\mathrm{mod}\ 16)$. Можно также считать, что $\bar{3} = -5$, поскольку $3 \cdot (-5) = -15 = 1 (\mathrm{mod}\ 16)$. Аналогично, $\overline{-3} = 5$, поскольку $(-3) \cdot 5 = -15 \equiv 1 (\mathrm{mod}\ 16)$.

Важность этих наблюдений заключается в том, что они показывают применимость рассматриваемых концепций как к знаковым, так и к беззнаковым числам. Если вы будете работать с беззнаковыми числами на 4-битовой машине, то получите $\bar{3} = 11$; если же

будете иметь дело со знаковыми числами, то получите $\bar{3} = -5$. Однако и 11 и -5 имеют одно и то же представление в дополнительном коде (поскольку их разность равна 16), так что в обоих случаях в качестве мультипликативного обратного элемента используется одно и то же содержимое машинного слова.

Рассмотренная теорема непосредственно применима к задаче деления (знакового и беззнакового) на нечетное число d на W-битовом компьютере. Поскольку любое нечетное число является взаимно простым с 2^W, теорема гласит, что если d нечетно, то существует целое $\bar{d}$ (единственное в диапазоне от 0 до $2^W - 1$ или от -2^{W-1} до $2^{W-1} - 1$), такое, что $d\bar{d} \equiv 1 (\bmod 2^W)$. Следовательно, для любого и, являющегося делителем d, справедливо соотношение

$$\frac{n}{d} \equiv \frac{n}{d}\left(d\bar{d}\right) \equiv n\bar{d}\left(\bmod 2^W\right).$$

Другими словами, n/d можно вычислить путем умножения и на $\bar{d}$, после чего взять правые W битполученного произведения.

Если делитель d представляет собой четное число, то пусть $d = d_0 \cdot 2^k$, где d_0 нечетно, а $k > 1$. Тогда просто выполняется сдвиг числа и вправо на k позиций (устраняя нулевые биты), после чего выполняется умножение на $\bar{d}_0$ (сдвиг можно выполнить и после операции умножения).

Ниже приведен код для деления на 7 числа n, кратного 7. Этот код дает корректный результат как при знаковом, так и при беззнаковом делении.

```
li M,OxB6DB6DB7  Мультипликативное обратное, (5*2**32+1)/7
mul q,M,n        q = n/7
```

Вычисление мультипликативного обратного по алгоритму Евклида

Каким же образом можно вычислить мультипликативное обратное число? Стандартный метод — использование "расширенного алгоритма Евклида". Этот метод вкратце рассматривается далее в приложении к нашей основной задаче, а заинтересовавшимся читателям можно посоветовать обратиться к [49] и [39, раздел 4.5.2].

Пусть задан нечетный делитель d и нам требуется найти число x, такое, что

$$dx \equiv 1(\bmod m),$$

причем в нашей задаче $m = 2^W$ (где W— размер машинного слова). Это может быть выполнено, если будет решено в целых числах x и y (положительных, отрицательных, равных 0) уравнение

$$dx + my = 1.$$

Начнем с того, что сделаем d положительным, добавляя к нему достаточное количество раз число m (d и $d + km$ имеют одно и то же мультипликативное обратное). Затем запишем следующие уравнения (где $d, m > 0$):

$$d(-1) + m(1) = m - d, \tag{31,а}$$

$$d(1) + m(0) = d. \tag{31,б}$$

Если $d = 1$, поставленная задача решена, так как (31,б) показывает, что $x = 1$. В противном случае вычислим

$$q = \left\lfloor \frac{m-d}{d} \right\rfloor.$$

Далее умножим уравнение (31,б) на q и вычтем его из (31,а). Это даст нам уравнение

$$d(-l-q) + \text{от}(1) = \text{от} - d - qd - \text{rem}(/\text{n} - d, d).$$

Это уравнение справедливо, поскольку мы просто умножили одно из уравнений на константу и вычли его из другого. Если $\text{rem}(m-d,d) = 1$, то задача решена (последнее уравнение и является решением) и $x = -1 - q$.

Повторим описанный процесс с последними двумя уравнениями, получая новое уравнение. Будем продолжать эти действия до тех пор, пока в правой части уравнения не будет получена 1. Тогда множитель при *d,* приведенный по модулю *m,* и будет представлять собой искомое мультипликативное обратное к *d.*

Если $m - d < d$, так что первое частное равно 0, то третья строка будет точной копией первой, так что второе частное будет ненулевым. Кроме того, в разных источниках зачастую в качестве первой строки используется следующая:

$$d(0) + m(1) = m,$$

но в нашем приложении $m = 2^w$ невозможно представить в компьютере.

Лучше всего проиллюстрировать описываемый процесс на конкретном примере. Пусть $m=256$ и $d=7$. Тогда процесс вычислений выглядит как показано ниже (для получения третьей строки используется $q = [249/7J = 35$).

```
 7(-1)  +  256( 1)  = 249
 7( 1)  +  256( 0)  = 7
7(-36)  +  256( 1)  = 4
7( 37)  +  256(-1)  = 3
7(-73)  +  256( 2)  = 1
```

Таким образом, мультипликативным обратным по модулю 256 числа 7 является -73 (или, используя числа из диапазона от 0 до 255, число 183). Проверяем: $7 \cdot 183 = 1281 \equiv 1 \pmod{256}$.

Начиная с третьей строки, числа в правом столбце представляют собой остатки от деления чисел, расположенных выше, так что это строго убывающая последовательность неотрицательных чисел, которая, таким образом, должна завершиться 0 (в примере выше для этого требуется еще один шаг). Кроме того, число перед 0 должно быть 1 по следующей причине. Предположим, что последовательность остатков завершается некоторым числом b, не равным 1, после которого идет число 0. Тогда целое, предшествующее b, должно быть кратным b (скажем, числом $k_1 b$) с тем, чтобы следующим остатком в последовательности было число 0. Такие же рассуждения приводят к выводу, что для того, чтобы получить остаток b, предшествующим $k_1 b$ числом должно быть $k_1 k_2 b + b$. Продолжая эту последовательность, вы увидите, что каждое из этих чисел должно быть кратно b, включая числа из первых двух строк, которые равны $m - d$ и d и являются взаимно простыми.

Приведенное выше рассуждение является неформальным доказательством того, что рассматриваемый процесс завершается, причем наличием 1 в правом столбце, а следовательно, находит мультипликативное обратное d число.

Для проведения данных вычислений на компьютере заметим, что если $d < 0$, то к нему необходимо прибавить 2^W. Однако в случае арифметики, использующей дополнительный к 2 код, это делать не обязательно; достаточно просто рассматривать d как беззнаковое число, независимо от того, как именно его рассматривает приложение.

Вычисление q должно использовать беззнаковое деление.

Заметим, что все вычисления можно выполнять по модулю m, так как это никак не влияет на правый столбец (его значения все равно остаются в диапазоне от 0 до $m-1$). Это важно, так как позволяет нам выполнять вычисления с "одинарной точностью", используя беззнаковую компьютерную арифметику по модулю 2^W.

Большинство величин в рассматриваемой таблице не обязательно должны быть представлены при вычислениях, например столбец множителей при 256, поскольку при решении уравнения $dx+my=\backslash$ нас не интересует значение y. Нет необходимости и в представлении d в первом столбце. Оставляя в таблице только необходимое, получим ее сокращенный вид.

255	249
1	7
220	4
37	3
183	1

Программа на языке С для проведения этих вычислений представлена в листинге 10.4.

Листинг 10.4. Алгоритм Евклида поиска мультипликативного обратного по модулю 2^{32}

```
unsigned mulinv(unsigned d)   // d должно быть нечетно
{
    unsigned x1, v1, x2, v2, x3, v3, q;
    x1 = 0xFFFFFFFF; vl = -d;
    X2 = 1; V2 = d;
    while (v2 > 1)
    {
        q = v1/v2;
        x3 = xl - q*x2; v3 = vl - q*v2;
        xl = x2; vl = v2;
        X2 = x3; V2 = v3;
    }
    return x2;
}
```

Причина использования условия цикла v2 > 1 вместо более естественного v2 != 1 заключается в том, что если при использовании последнего условия функции будет ошибочно передано четное значение, то цикл может никогда не завершиться (если аргумент d четен, v2 никогда не получит значение 1, но в конечном итоге примет значение 0).

Что же вычисляет данная программа, если передать ей четный аргумент? Как и следует из формул, она вычисляет число x, такое, что $dx \& \text{o}(\text{mod } 2^{32})$, которое вряд ли может оказаться полезным. Однако минимальные изменения в программе (изменение условия цикла на v2 != о и возврат значения xl вместо x2) приводят к вычислению ею числа x,

такого, что $dx \equiv g \pmod{2^{32}}$, где g — наибольший общий делитель d и 2^{32}, т.е. наибольшей степени 2, являющейся делителем d. Такая модифицированная программа, как и ранее, вычисляет для нечетного d мультипликативное обратное число, хотя и требует для этого на одну итерацию больше.

Что касается количества итераций, выполняемых данной программой, то для нечетного d, не превышающего 20, требуется максимум три итерации; среднее их число равно 1.7. Для d порядка 1000 требуется максимум 11, а в среднем — шесть итераций.

Вычисление мультипликативного обратного по методу Ньютона

Хорошо известно, что при работе с действительными числами значение $1/d$, где $d \neq 0$, может быть вычислено с возрастающей точностью с помощью итеративного вычисления

$$x_{n+1} = x_n(2 - dx_n) \qquad (32)$$

при начальном приближении x_0, достаточно близком к $1/d$. Количество точных цифр в результате при каждой итерации примерно удваивается.

Однако гораздо менее известно, что данная формула может применяться для поиска мультипликативного обратного числа при использовании модульной арифметики с целыми числами! Например, для поиска мультипликативного обратного к 3 по модулю 256 начнем с $x_1 = 1$ (можно использовать любое нечетное число). Тогда

$$\begin{aligned}
x_1 &= 1(2 - 3 \cdot 1) = -1, \\
x_2 &= -1(2 - 3(-1)) = -5, \\
x_3 &= -5(2 - 3(-5)) = -85, \\
x_4 &= -85(2 - 3(-85)) = -21845 = -85 \pmod{256}.
\end{aligned}$$

Итерации достигли неподвижной точки по модулю 256, так что -85, или 171, и есть мультипликативное обратное числа 3 по модулю 256. Все приведенные вычисления выполнялись по модулю 256.

Почему работает этот метод? Поскольку если x_n удовлетворяет условию

$$dx_n = 1 \pmod m$$

и x_{n+1} определяется формулой (32), то

$$dx_{n+1} \equiv 1 \pmod{m^2}.$$

Чтобы увидеть это, положим $dx_n = 1 + km$. Тогда

$$\begin{aligned}
dx_{n+1} &= dx_n(2 - dx_n) = \\
&= (1 + km)(2 - (1 + km)) = \\
&= (1 + km)(1 - km) = \\
&= 1 - k^2m^2 = \\
&= 1 \pmod{m^2}.
\end{aligned}$$

В нашем приложении m является степенью 2, скажем, 2^N. В этом случае, если

$$dx_n \equiv 1 \pmod{2^N}, \text{ то}$$
$$dx_{n+1} \equiv 1 \pmod{2^{2N}}.$$

В этом смысле, если x_n рассматривается как некоторое приближение $\bar{d}$, то каждая итерация (32) удваивает количество битов "точности" приближения.

Так случилось, что по модулю 8 мультипликативным обратным любого нечетного числа d является само это число. Таким образом, в качестве начального приближения можно взять $x_0 = d$. Тогда по формуле (32) получим значения $x_1, x_2, \ldots$, такие, что

$$dx_1 \equiv 1 \pmod{2^6},$$
$$dx_2 \equiv 1 \pmod{2^{12}},$$
$$dx_3 \equiv 1 \pmod{2^{24}},$$
$$dx_4 \equiv 1 \pmod{2^{48}} \text{ и т.д.}$$

Таким образом, четырех итераций достаточно, чтобы найти мультипликативное обратное по модулю 2^{32} (если $x \equiv 1 \pmod{2^{48}}$, то $x = 1 \pmod{2^n}$ для $« \le 48$). Это приводит нас к программе на языке С (листинг 10.5), где все вычисления выполняются по модулю 2^{32}.

Листинг 10.5. Вычисление мультипликативного обратного по модулю 2^{32} методом Ньютона

```
unsigned mulinv (unsigned d)   // d должно быть нечетно
{
    unsigned xn, t;
    xn = d;
loop:
    t = d*xn;
    if (t == 1) return xn;
    xn = xn*(2 - t);
    goto loop;
}
```

Примерно для половины значений d этой программе требуется 4.5 итерации, т.е. девять умножений. Для другой половины (у которой "верны четыре бита" начального значения xn, т.е. $d^2 \equiv 1 \pmod{16}$) требуется, как правило, семь (или меньшее количество) умножений. Таким образом, можно считать, что этот метод требует в среднем восьми умножений.

Один из вариантов этой программы состоит в простом выполнении цикла четыре раза, независимо от значения d, что позволяет обойтись восемью умножениями и без команд управления циклом. Другой вариант предусматривает выбор в качестве начального приближения мультипликативного обратного "с точностью 4 бита", т.е. перед началом вычислений нужно найти x_0, которое удовлетворяет условию $dx_0 \equiv 1 \pmod{16}$. Тогда для вычислений потребуется выполнить только три итерации цикла. Найти подходящее начальное приближение можно, например, следующими способами:

$$x_0 \leftarrow d + 2((d+1) \,\&\, 4) \text{ или}$$
$$x_0 \leftarrow d^2 + d - 1.$$

Здесь умножение на 2 выполняется при помощи сдвига влево, а все вычисления производятся по модулю 2^{32} (игнорируя переполнения). Поскольку во второй формуле используется умножение, ее применение позволяет сэкономить только одну такую команду.

Такое пристальное рассмотрение вопроса времени вычислений, конечно, не имеет смысла, если метод применяется в компиляторе. В этом случае данная программа используется настолько редко, что при ее кодировании в первую очередь должен интересовать ее размер. Тем не менее могут существовать приложения, где поиск мультипликативного обратного должен выполняться максимально быстро.

Некоторые мультипликативные обратные

В завершение перечислим в табл. 10.3 некоторые мультипликативные обратные числа.

Таблица 10.3. Некоторые мультипликативные обратные числа

d	$\bar{d}$		
(десятичное)	mod 16 (десятичное)	mod 2^{32} (шестнадцатеричное)	mod 2^{64} (шестнадцатеричное)
-7	-7	49249249	92492492 49249249
-5	3	33333333	33333333 33333333
-3	5	55555555	55555555 55555555
-1	-1	FPFFFFFF	FFFFFFFF FFFFFFFF
1	1	1	1
3	11	AAAAAAAB	AAAAAAAA AAAAAAAB
5	13	CCCCCCCD	CCCCCCCC CCCCCCCD
7	7	B6DB6DB7	6DB6DB6D B6DB6DB7
9	9	38E38E39	8E38E38E 38E38E39
11	3	BA2E8BA3	2E8BA2E8 BA2E8BA3
13	5	C4EC4EC5	4EC4EC4E C4EC4EC5
15	15	EEEEEEEF	EEEEEEEE EEEEEEEF
25		C28F5C29	8F5C28F5 C28F5C29
125		26E978D5	1CAC0831 26E978D5
625		3AFB7E91	D288CE70 3AFB7E91

Вы можете заметить, что в ряде случаев (d=3,5,9,11) мультипликативное обратное к d совпадает с магическим числом для беззнакового деления на d (см. раздел 10.14). Это в большей или меньшей мере случайность. Подобное происходит для тех чисел, для которых магическое число M равно множителю m, и эти величины имеют вид $(2^p+1)/d$ при $p>32$. В этом случае

$$Md = \left(\frac{2^p+1}{d}\right)d \equiv 1 \left(\mathrm{mod}\ 2^{32}\right),$$

так что $M s d \left(\mathrm{mod}\ 2^{32}\right)$.

10.16. Проверка нулевого остатка при делении на константу

Мультипликативное обратное делителя d может использоваться для проверки равенства 0 остатка после деления на d [21].

Беззнаковое деление

Сначала рассмотрим беззнаковое деление на нечетный делитель d. Обозначим через d мультипликативное обратное к d число. Тогда, поскольку $dd \equiv 1\left(\mathrm{mod}\ 2^W\right)$, где W — размер машинного слова в битах, $\bar{d}$ также нечетно. Следовательно, $\bar{d}$ является взаимно простым с 2^W и, как показано в доказательстве теоремы MI в предыдущем разделе, если n представляет собой множество всех 2^W различных чисел по модулю 2^W, то rid также представляет собой множество всех 2^W различных чисел по модулю 2^W.

В предыдущем разделе было показано, что если n кратно d, то

$$\frac{n}{d} \equiv n\bar{d} \pmod{2^W}.$$

Следовательно, $n\bar{d} = 0, 1, 2, \ldots, \lfloor (2^W - 1)/d \rfloor \pmod{2^W}$ для $n = 0, d, 2d, \ldots, \lfloor (2^W - 1)/d \rfloor d$. Таким образом, для n, *не* являющегося кратным d, значение $n\bar{d}$, приведенное по модулю 2^W к диапазону от 0 до $2^W - 1$, должно превышать величину $\lfloor (2^W - 1)/d \rfloor$.

Этот вывод можно использовать для проверки на равенство нулю остатка от деления. Например, чтобы проверить, что число и кратно 25, умножим n на $\overline{25}$ и сравним правые W бит со значением $\lfloor (2^W - 1)/25 \rfloor$. Использование базового набора RISC-команд дает следующий код:

```
li      M,0xC28F5C29  Загружаем мультипликативное обратное 25
mul     q,M,n          q = правая половина M*п
li      c,0x0A3D70A3  C = floor((2**32-1)/25)
cmpleu  t,q,c         Сравниваем q и c; переход,
bt      t,is_mult      если п кратно 25
```

Для того чтобы распространить этот метод на четные делители, представим делитель в следующем виде: $d = d_0 \cdot 2^k$, где d_0 нечетно, а $k \ge 1$. Тогда, поскольку целое число делится на d тогда и только тогда, когда оно делится на d_0 и 2^k, и поскольку n и $n\bar{d}_0$ имеют одинаковое количество завершающих нулевых битов ($\bar{d}_0$ нечетно), проверка того, что n кратно d, состоит в следующем:

$$q = \operatorname{mod}(n\bar{d}_0, 2^W);$$

$q \le \lfloor (2^W - 1)/d_0 \rfloor$ и q завершается не менее чем k нулевыми битами,

где под функцией "mod" подразумевается приведение $n\bar{d}_0$ к интервалу $[0, 2^W - 1]$.

Непосредственная реализация описанного метода требует двух проверок и условных переходов, однако если компьютер имеет команду *циклического сдвига,* то метод можно реализовать с помощью одного *сравнения с ветвлением.* Это следует из приведенной далее теоремы, в которой запись $a \overset{rot}{\gg} k$ означает компьютерное слово а, циклически сдвинутое вправо на k бит ($0 \le k < 32$).

Теорема ZRU. *$x \overset{u}{\le} a$ и x заканчивается k нулевыми битами тогда и только тогда, когда $x \overset{rot}{\gg} k \overset{u}{\le} \lfloor a/2^k \rfloor$.*

Доказательство. (Будем считать, что имеем дело с 32-разрядным компьютером.) Предположим, что $x \overset{u}{\le} a$ и x заканчивается k нулевыми битами. Тогда, так как $x \overset{u}{\le} a$, то $\lfloor x/2^k \rfloor \overset{u}{\le} \lfloor a/2^k \rfloor$. Однако $\lfloor x/2^k \rfloor = x \overset{rot}{\gg} k$ и, таким образом, $x \overset{rot}{\gg} k \overset{u}{\le} \lfloor a/2^k \rfloor$. Если x не заканчивается k нулевыми битами, то $x \overset{rot}{\gg} k$ не начинается с k нулевых битов, в то время как $\lfloor a/2^k \rfloor$ начинается с них, так что $x \overset{rot}{\gg} k \overset{u}{>} \lfloor a/2^k \rfloor$. И наконец, если $x \overset{u}{>} a$ и x заканчивается k

нулевыми битами, то целое число, образованное первыми $32-k$ битами x должно превышать число, образованное первыми $32-k$ битами a, так что $\lfloor x/2^k \rfloor \overset{u}{>} \lfloor a/2^k \rfloor$.

При использовании этой теоремы проверка того, что n кратно d, где n и d — ненулевые беззнаковые целые числа, причем $d = d_0 \cdot 2^k$, где d_0 — нечетно, выполняется следующим образом:

$$q \leftarrow \mathrm{mod}\left(n\bar{d}_0, 2^W\right); \\ q \overset{rot}{\gg} k \overset{u}{\le} \left\lfloor \left(2^W - 1\right)/d \right\rfloor.$$

Здесь мы воспользовались тем, что $\left\lfloor \left\lfloor \left(2^W - 1\right)/d_0 \right\rfloor /2^k \right\rfloor = \left\lfloor \left(2^W - 1\right)/\left(d_0 \cdot 2^k\right) \right\rfloor = \left\lfloor \left(2^W - 1\right)/d \right\rfloor$.

Далее в качестве примера приведен код, проверяющий, является ли беззнаковое целое число n кратным 100.

```
li     M,0xC28F5C29   Мультипликативное обратное 25
mul    q,M,n          q = правая половина M*n
shrri  q,q,2          Циклический сдвиг вправо на два бита
li     c,x'028F5C28'  c = floor((2**32-1)/100)
cmpleu t,q,c          Сравнение q и c, и переход, если
bt     t,is_mult       n кратно 100
```

Знаковое деление, делитель ≥2

В случае знакового деления, как было показано в предыдущем разделе, если n кратно d и d нечетно, то

$$\frac{n}{d} \equiv n\bar{d} \left(\mathrm{mod}\ 2^W\right).$$

Таким образом, для $n = \left\lceil -2^{W-1}/d \right\rceil \cdot d, \ldots -d, 0, d, \cdots \left\lfloor \left(2^{W-1} - 1\right)/d \right\rfloor \cdot d$ имеем: $nd \equiv \left\lceil -2^{W-1}/d \right\rceil, \ldots -1, 0, 1, \cdots \left\lfloor \left(2^{W-1} - 1\right)/d \right\rfloor \left(\mathrm{mod}\ 2^W\right)$. Кроме того, поскольку d взаимно простое с 2^W, то если n принимает 2^W различных значений по модулю 2^W, тогда nd также принимает 2^W различных значений по модулю 2^W. Следовательно, n кратно d тогда и только тогда, когда

$$\left\lceil -2^{W-1}/d \right\rceil \le \mathrm{mod}\left(n\bar{d}, 2^W\right) \le \left\lfloor \left(2^{W-1} - 1\right)/d \right\rfloor,$$

где под функцией "mod" подразумевается приведение nd к интервалу $\left[-2^{W-1}, 2^{W-1} - 1\right]$.

Этот способ можно немного упростить, заметив, что, так как d нечетно, а из наших начальных предположений оно положительно и не равно единице, d не является делителем 2^{W-1}. Таким образом,

$$\left\lceil -2^{W-1}/d \right\rceil = \left\lceil \left(-2^{W-1} + 1\right)/d \right\rceil = -\left\lfloor \left(2^{W-1} - 1\right)/d \right\rfloor.$$

Для знаковых чисел проверка того, что n является кратным d, где $d = d_0 \cdot 2^k$, а d_0 нечетно, выполняется следующим образом:

$$q = \mathrm{mod}\left(n\bar{d}_0, 2^W\right);$$

$-\left\lfloor \left(2^{W-1} - 1\right)/d_0 \right\rfloor \le q \le \left\lfloor \left(2^W - 1\right)/d_0 \right\rfloor$ и q завершается не менее чем k нулевыми битами.

На первый взгляд кажется, что при таком методе требуется три проверки и перехода. Однако, как и в беззнаковом случае, все можно свести к одной команде *сравнения с ветвлением,* если воспользоваться следующей теоремой.

Теорема ZRS. *Если* $a \ge 0$, *то следующие утверждения эквивалентны:*

1) $-a \le x \le a$ *и x заканчивается* k *или большим количеством нулевых битов,*

2) $\text{abs}(x) \overset{rot}{\gg} k \overset{u}{\le} \lfloor a/2^k \rfloor$,

3) $x + a' \overset{rot}{\gg} k \overset{u}{\le} \lfloor 2a'/2^k \rfloor$,

где a' *представляет собой a с установленными в 0 правыми k битами (т.е.* $a' = a \,\&\, -2^k$ *).*

Доказательство. (Будем считать, что имеем дело с 32-разрядным компьютером.) Чтобы убедиться, что утверждение 1 эквивалентно утверждению 2, достаточно заметить, что выражения $-a \le x \le a$ и $\text{abs}(x) \le a$ эквивалентны. После этого эквивалентность утверждений 1 и 2 следует из теоремы ZRU.

Чтобы убедиться в эквивалентности утверждений 1 и 3, заметим, что утверждение 1 справедливо и при замене a на a'. Тогда, в соответствии с теоремой границ из раздела 4.1, оно, в свою очередь, эквивалентно

$$x + a' \overset{u}{\le} 2a'.$$

Поскольку $x + a$ заканчивается k нулевыми битами тогда и только тогда, когда k нулевыми битами заканчивается x, можно применить теорему ZRU, которая и дает требующийся нам результат.

Используя утверждение 3 рассмотренной теоремы, проверку кратности n числу d, где л и d — знаковые целые числа, не меньшие 2, и $d = d_0 \cdot 2^k$, где d_0 нечетно, можно провести следующим образом:

$$q \leftarrow \text{mod}(n\bar{d}_0, 2^W);$$
$$a' \leftarrow \lfloor (2^{W-1} - 1)/d_0 \rfloor \,\&\, -2^k;$$
$$q + a' \overset{rot}{\gg} k \overset{u}{\le} \lfloor (2a')/2^k \rfloor.$$

(Поскольку d — константа, a' можно вычислить во время компиляции.)

Далее в качестве примера приведен код, проверяющий, является ли знаковое целое число и кратным 100. Обратите внимание, что константа $\lfloor 2a'/2^k \rfloor$ в любой момент может быть получена из константы a' путем сдвига на $k-1$ позиций, что позволяет сохранить одну команду либо загрузку из памяти для получения операнда сравнения.

```
li      M,OXC28F5C29  Загрузка мультипликативного обратного 25
mul     q.M,n         q = правая половина M*п
li      c,OX051EB850  c = floor((2**31 - 1)/25) & -4
add     q,q,c         Прибавляем с
shrri   q.q,2         Циклический сдвиг вправо на 2 позиции
shri    c,c,1         Вычисление константы для сравнения
cmpleu  t,q,c        Сравнение q и с, и переход, если
bt      t,is_mult    n кратно 100
```

ГЛАВА И
НЕКОТОРЫЕ ЭЛЕМЕНТАРНЫЕ ФУНКЦИИ

11.1. Целочисленный квадратный корень

Под "целочисленным квадратным корнем" подразумевается функция $\lfloor \sqrt{x} \rfloor$. Чтобы расширить область применения и чтобы избежать решения вопроса о том, как поступать с отрицательными аргументами функции, будем считать x беззнаковой величиной: $0 < x < 2^{32} - 1$.

Метод Ньютона

Для чисел с плавающей точкой, по сути, универсальным методом вычисления квадратного корня является метод Ньютона. Этот метод начинается с некоего значения g_0, которое является начальной оценкой $\sqrt{a}$. Затем выполняется серия уточнений значения квадратного корня по формуле

$$g_{n+1} = \left(g_n + \frac{a}{g_n} \right) \Big/ 2 .$$

Приведенный метод имеет квадратичную сходимость, т.е. если в некоторый момент g_n имеет точность n битов, то g_{n+1} имеет точность $2n$ битов. Рабочая программа должна иметь средства для определения достижения необходимой точности и прекращения вычислений.

Приятной неожиданностью оказывается то, что метод Ньютона хорошо работает и в области целых чисел. Для того чтобы убедиться в этом, рассмотрим следующую теорему.

Теорема. *Пусть* $g_{n+1} = \lfloor (g_n + \lfloor a/g_n \rfloor)/2 \rfloor$, *где* g_n *— целое число, большее 0. Тогда*

а) если $g_n > \lfloor \sqrt{a} \rfloor$, *то* $\lfloor \sqrt{a} \rfloor \le g_{n+1} < g_n$;

б) если $g_n = \lfloor \sqrt{a} \rfloor$, ***то*** $\lfloor \sqrt{a} \rfloor \le g_{n+1} \le \lfloor \sqrt{a} \rfloor + 1$.

Иными словами, если у нас есть завышенная целочисленная оценка g_n величины $\lfloor \sqrt{a} \rfloor$, то следующее приближение будет строго меньше предыдущего, но не меньше $\lfloor \sqrt{a} \rfloor$. Таким образом, если начать со слишком большого приближения, то будет получена монотонно убывающая последовательность приближений. Если же начальное приближение $g_n = \lfloor \sqrt{a} \rfloor$, то следующее приближение оказывается либо равным предыдущему, либо на 1 большим. Таким образом, мы получаем способ определения того, что последовательность приближений сошлась: если начать с $g_0 > \lfloor \sqrt{a} \rfloor$, то точное решение g_n будет достигнуто в тот момент, когда $g_{n+1} \ge g_n$.

Случай $a=0$ следует рассматривать отдельно в связи с тем, что при этом может возникнуть ситуация деления 0 на 0.

Доказательство. а) Поскольку g_n – целое число,

$$g_{n+1} = \left\lfloor \left(g_n + \left\lfloor \frac{a}{g_n} \right\rfloor \right) \Big/ 2 \right\rfloor = \left\lfloor \left\lfloor g_n + \frac{a}{g_n} \right\rfloor \Big/ 2 \right\rfloor = \left\lfloor \left(g_n + \frac{a}{8_{,i}} \right) \Big/ 2 \right\rfloor = \left\lfloor \frac{g_n^2 + a}{2g_n} \right\rfloor.$$

Так как $g_n > \lfloor \sqrt{a} \rfloor$ и g_n— целое число, $g_n > \sqrt{a}$. Определим е следующим образом: $g_n = (1+\varepsilon)\sqrt{a}$. Тогда е > 0 и

$$\left\lfloor \frac{g_n^2 + a}{2g_n} \right\rfloor = g_{n+1} \le \frac{g_n^2 + a}{2g_n},$$

$$\left\lfloor \frac{(1+\varepsilon)^2 a + a}{2(1+\varepsilon)\sqrt{a}} \right\rfloor = g_{n+1} < \frac{g_n^2 + g_n^2}{2g_n},$$

$$\left\lfloor \frac{2+2\varepsilon+\varepsilon^2}{2(1+\varepsilon)} \sqrt{a} \right\rfloor = g_{n+1} < g_n,$$

$$\left\lfloor \frac{2+2\varepsilon}{2(1+\varepsilon)} \sqrt{a} \right\rfloor \le g_{n+1} < g_n,$$

$$\lfloor \sqrt{a} \rfloor \le g_{n+1} < g_n.$$

б) Поскольку $g_n = \lfloor \sqrt{a} \rfloor$, $\sqrt{a} - 1 < g_n < \sqrt{a}$, так что $g_n^2 < a < (g_n + 1)^2$. Следовательно,

$$\left\lfloor \frac{g_n^2 + g_n^2}{2g_n} \right\rfloor \le g_{n+1} \le \left\lfloor \frac{g_n^2 + (g_n + 1)^2}{2*.} \right\rfloor,$$

$$\lfloor g_n \rfloor \le g_{n+1} \le \left\lfloor g_n + 1 + \frac{1}{2*.} \right\rfloor,$$

$$\lfloor \sqrt{a} \rfloor < g_{n+1} < \lfloor g_n + 1 \rfloor \text{ (т.к. } g_n \text{ целое и } \frac{1}{2g_n} < 1\text{)},$$

$$\lfloor \sqrt{a} \rfloor \le g_{n+1} \le \lfloor g_n \rfloor + 1 = \lfloor \sqrt{a} \rfloor + 1.$$

Сложной частью при использовании метода Ньютона для вычисления $\lfloor \sqrt{x} \rfloor$ является получение первого приближения. Процедура в листинге 11.1 устанавливает первое приближение g_0 равным наименьшей степени 2, которая больше или равна $\sqrt{x}$. Например, $g_0=2$ для $x=4$, а для $x=5$ в качестве первого приближения принимается $g_0=4$.

Листинг 11.1. Метод Ньютона поиска целочисленного квадратного корня

```
int isqrt(unsigned x)
{
   unsigned x1;
   int s, gO, g1;
   if (x <= 1) return x;
   s = 1;
   xl = x - 1;
   if (Xl > 65535)  s = s + 8; Xl = Xl >> 16;}
   if (Xl > 255)    S = s + 4; Xl = Xl >> 8;}
   if (Xl > 15)     S = S + 2; Xl = Xl >> 4;}
   if (Xl > 3)      s = s + 1;}
   gO = 1 << s;    // gO = 2**s
```

```
    g1 = (g0 + (x >> s)) >> 1;   // g1 = (g0 + x/g0)/2
    while (g1 < g0) { // Повторяем, пока приближения
        g0 = g1;      // строго уменьшаются
        g1 = (g0 + (x/g0)) >> 1;
    }
    return g0;
}
```

Поскольку первое приближение g_0 представляет собой степень 2, для получения g_1 нет необходимости в реальном делении — можно обойтись сдвигом вправо.

Поскольку точность первого приближения составляет около 1 бита, а метод Ньютона обеспечивает квадратичную сходимость (число точных битов удваивается при каждой итерации), следует ожидать, что процедура поиска квадратного корня завершится за пять итераций (на 32-битовом компьютере), для чего потребуется четыре деления (первое деление заменяется сдвигом вправо). Эксперименты показывают, что максимальное количество делений равно пяти (четырем для аргументов, меньших 16785407).

Если в компьютере имеется команда вычисления количества ведущих нулевых битов, то получить первое приближение очень легко: заменить первые семь выполняемых строк в приведенной выше процедуре строками

```
if (x <= 1) return x;
s = 16 - nlz(x - 1)/2;
```

Еще один вариант в случае отсутствия команды вычисления количества ведущих нулевых битов — вычисление s посредством бинарного дерева поиска. Этот метод позволяет получить несколько лучшее приближение g_0: наименьшую степень 2, которая больше или равна $\lfloor \sqrt{x} \rfloor$. Для некоторых значений x это дает меньшее значение g_0, но достаточно большое, чтобы выполнялся критерий сходимости из рассмотренной ранее теоремы. Отличие этих схем показано ниже.

Диапазон x для листинга 11.1	Диапазон x для листинга 11.2	Первое приближение
0	0	0
1	От 1 до 3	1
От 2 до 4	От 4 до 8	2
От 5 до 16	От 9 до 24	4
От 17 до 64	От 25 до 80	8
От 65 до 256	От 81 до 288	16
...	...	...
От $2^{28}+1$ до 2^{30}	От $(2^{14}+1)^2$ до $(2^{15}+1)^2-1$	215
От $2^{30}+1$ до $2^{32}-1$	От $(2^{15}+1)^2$ до $2^{32}-1$	216

Соответствующая процедура показана в листинге 11.2. Она особенно удобна при работе с малыми значениями x ($0 \le x \le 24$), так как при этом не требуется выполнение деления.

Листинг 11.2. Целочисленный квадратный корень с вычислением первого приближения путем бинарного поиска

```
int isqrt(unsigned x)
{
    int s, g0, g1;
    if (x <= 4224)
```

```
    if (X <= 24)
        if (x <= 3) return (x + 3) >>2;
        else if (x <= 8) return 2;
        else return (x >> 4) + 3;
    else if (x <= 288)
        if (x <= 80)    s = 3; else s = 4;
    else if (x <= 1088) x = 5; else s = 6;
else if (x <= 1025*1025 - 1)
    if (x <= 257*257 - 1)
        if (x <= 129*129 - 1)  S = 7; else s = 8;
    else if (x <= 513*513 - 1) s = 9; else s = 10;
else if (x <= 4097*4097 - 1)
    if (X <= 2049*2049 - 1) s = 11; else s = 12;
else if (x <= 16385*16385 - 1)
    if (x <= 8193*8193 - 1)   S = 13; else s = 14;
else if (x <= 32769*32769 - 1) s = 15; else s = 16;
gO = 1 << s; // gO = 2**s
// Далее все так же, как и в листинге 11.1
```

Время выполнения алгоритма из листинга 11.1 на компьютере с базовым набором RISC-команд в наихудшем случае составляет около $26+(D+6)n$ тактов, где D — время выполнения команды деления в циклах, а n — количество итераций цикла `while`. Время выполнения алгоритма из листинга 11.2 составляет в наихудшем случае $27+(D+6)n$ тактов в предположении (в обоих случаях), что команда *ветвления* выполняется за один такт. В приведенной ниже таблице показаны среднее количество итераций цикла в обоих алгоритмах для равномерно распределенного в указанных диапазонах значения *x*.

x	Листинг 11.1	Листинг 11.2
От 0 до 9	0.80	0
От 0 до 99	1.46	0.83
От 0 до 999	1.58	1.44
От 0 до 9999	2.13	2.06
От 0 до $2^{32}-1$	2.97	2.97

Если считать, что время деления равно 20 тактам, а *x* равномерно распределено от О до 9999, то оба алгоритма требуют около 81 такта процессорного времени.

Бинарный поиск

Поскольку алгоритм, основанный на методе Ньютона, начинается с бинарного поиска начального приближения, почему бы не воспользоваться бинарным поиском самого квадратного корня? Этот способ может начать работу с двух границ: по всей видимости, О и 2^{16}. Затем в качестве приближения квадратного корня берется середина между границами. Если квадрат этого значения больше аргумента *x,* то средняя точка становится новой верхней границей; в противном случае средняя точка становится новой нижней границей. Итерации продолжаются до тех пор, пока нижняя и верхняя границы не будут отличаться на 1. При этом нижняя граница будет представлять собой искомый результат.

Таким образом можно избежать деления, но потребуется значительное количество умножений — 16, если в качестве начальных границ воспользоваться значениями 0 и 2^{16}. (Этот метод на каждой итерации вычисляет по одному биту точного значения.) В листин-

ге 11.3 показана одна из вариаций данного алгоритма, в которой используются несколько улучшенные по сравнению со значениями 0 и 2^{16} начальные границы. Кроме того, процедура в листинге 11.3 для большинства RISC-компьютеров экономит один такт на цикл, изменяя a и b так, что в программе используется сравнение $b \geq a$ вместо $b - a \geq 1$.

Листинг 11.3. Бинарный поиск целочисленного квадратного корня

```
int isqrt(unsigned x)
{
    unsigned a, b, m; // Границы и средняя точка
    a = 1;
    b = (x >> 5) + 8; // См. пояснения в тексте
    if (b > 65535) b = 65535;
    do {
        m = (a + b) >> 1;
        if (m*m > x) b = m - 1;
        else a = m + 1;
    } while (b >= a);
    return a - 1;
}
```

В начале каждой итерации должны выполняться предикаты $a \leq \lfloor \sqrt{x} \rfloor + 1$ и $b \geq \lfloor \sqrt{x} \rfloor$. Начальное значение b должно быть легко вычислимым и близким к $\lfloor \sqrt{x} \rfloor$. Вот некоторые из возможных начальных значений: x, $x+4+1$, $x+8+2$, $x+16+4$, $x+32+8$, $x+64+16$ и т.д. Выражения в начале этого списка лучше подходят для небольших значений x, выражения ближе к концу списка — для x побольше. (Значение $x+2+1$ также применимо, но не имеет смысла, так как выражение $x+4+1$ везде дает лучшую или такую же границу.)

Семь вариаций приведенного в листинге 11.3 алгоритма можно получить, заменяя a на $a+1$, b на $b-1$, используя вместо $m=(a+b)+2$ выражение $m=(a+b+1)+2$ или комбинируя перечисленные подстановки.

Время выполнения процедуры из листинга 11.3 составляет примерно $6+(M+7.5)n$ тактов, где M — время выполнения умножения в тактах, а n — количество выполняемых итераций цикла. Приведенная ниже таблица содержит средние количества выполнения циклов для равномерно распределенных в указанных интервалах значений x.

x	Среднее количество итераций цикла
От 0 до 9	3.00
От 0 до 99	3.15
От 0 до 999	4.68
От 0 до 9999	7.04
От 0 до $2^{32}-1$	16.00

Если считать, что время выполнения умножения — 5 тактов, а x равномерно распределено от 0 до 9999, то время выполнения алгоритма составит около 94 тактов. Максимальное время выполнения ($n=16$) равно 206 тактам.

Если компьютер оснащен командой *подсчета ведущих нулевых битов*, то начальные границы можно определить следующим образом:

```
b = (1 << (33 - nlz(x))/2) - 1;
a = (b + 3)/2;
```

или $b = 2^{(32-\mathrm{nlz}(x)) \div 2} - 1$. Это очень хорошие границы для небольших x (так, при $0 \le x \le 15$ требуется только одна итерация), но для больших значений x дает только небольшое улучшение по сравнению с границами из листинга 11.3. При x из диапазона от 0 до 9999 среднее количество итераций примерно равно 5.45, что дает время выполнения около 74 тактов (при использовании тех же предположений, что и раньше).

Аппаратный алгоритм

Существует алгоритм, использующий при вычислении квадратного корня сдвиги и вычитания, который очень похож на алгоритм аппаратного деления из листинга 9.2. Будучи аппаратно встроен в 32-битовый компьютер, этот алгоритм использует 64-битовый регистр, инициализируемый аргументом x, за которым следует 32 нулевых бита. При каждой итерации 64-битовый регистр смещается на два бита влево, а текущий результат y (изначально равный 0) — на один бит. Затем из левой половины 64-битового регистра вычитается значение $2y+1$. Если результат вычитания неотрицателен, он заменяет левую половину 64-битового регистра, а к значению y прибавляется 1 (для этого не требуется сумматор, так как y в этот момент заканчивается нулевым битом). Если результат вычитания отрицателен, то и 64-битовый регистр, и y остаются неизменными. Итерации выполняются 16 раз.

Этот алгоритм был описан в 1945 году [36].

Пожалуй, неожиданным результатом оказывается то, что время работы этого процесса составляет примерно половину времени, необходимого для выполнения деления $64 \div 32 \Rightarrow 32$ при помощи упомянутого аппаратного алгоритма, так как в этом случае выполняется в два раза меньше итераций примерно той же сложности, что и в алгоритме деления.

При программном использовании данного алгоритма, вероятно, лучше всего избежать использования сдвига в двойном слове, которое требует около четырех команд сдвига. Алгоритм из листинга 11.4 [19] осуществляет это путем сдвига y и маскирования бита m справа. В среднем ему требуется выполнение 149 базовых RISC-команд. Два выражения y|m могут быть заменены сложением y+m.

Листинг 11.4. Аппаратный алгоритм поиска целочисленного квадратного корня

```
int isqrt(unsigned x)
{
    unsigned m, y, b;
    m = 0x40000000;
    y = 0;
    while (m != 0) { // Выполняется 16 раз
        b = y | m;
        y = y >> 1;
        if (x >= b) {
            x = x - b;
            y = y | m;
        }
        m = m >> 2;
    }
    return y;
}
```

Работа этого алгоритма похожа на школьный метод поиска квадратного корня. Ниже показан пример поиска $\lfloor\sqrt{179}\rfloor$ в соответствии с этим алгоритмом на 8-битовом компьютере.

```
 1011 OOH   xO Изначально, x = 179  (0xB3)
- 1         b1
 ---------
 0111 OOH   x1 0100 0000 y1
- 101       Ь2 0010 0000 y2
 ---------
 0010 OOH   X2 OOH 0000 y2
-  11 01    Ь3 0001 1000 y3
 ---------
 0010 OOH  x3 0001 1000 y3  (Вычитать нельзя)
-   1 1001  Ь4 0000 1100 y4
 ---------
 0000 1010  X4 0000 1101 y4
```

Результат вычислений равен 13; в регистре x остается остаток 10.

При использовании обычного трюка со *знаковым сдвигом вправо* на 31 бит можно избежать выполнения проверки if x>=b. Можно доказать, что старший бит b всегда нулевой (на самом деле $b < 5 \cdot 2^{28}$), что упрощает предикат x>=Ь (см. раздел 2.11). В результате группу операторов if можно заменить следующими:

```
t = (int)(x | ~(x - b)) >> 31; // -1, если x >= Ь, иначе O
x = x - (Ь & t);
y = y | (т & t);
```

Тем самым мы заменяем три такта семью в предположении, что компьютер оснащен командой ***или-не***, однако это может оказаться выгодной заменой, если условный переход в данном контексте требует более 5 тактов.

Представляется, что должен быть более простой метод программного вычисления целочисленного квадратного корня, чем требующий нескольких сотен тактов процессора. Ниже приведен ряд выражений для вычисления целочисленного квадратного корня для очень малых аргументов. Эти выражения могут оказаться полезными для ускорения рассмотренных алгоритмов, когда ожидается извлечение квадратного корня из малых чисел.

Выражение	Диапазон корректности	Количество RISC-команд (расширенный набор)
x	От 0 до 1	0
$x > 0$	От 0 до 3	1
$(x+3)\overset{u}{\div}4$	От 0 до 3	2
$x \overset{u}{\gg} \left(x \overset{u}{\div} 2\right)$	От 0 до 3	2
$x \overset{u}{\gg} (x > 1)$	От 0 до 5	2
$(x+12)\overset{u}{\div}8$	От 1 до 8	2
$(x+15)\overset{u}{\div}8$	От 4 до 15	2
$(x > 0) + (x > 3)$	От 0 до 8	3
$(x > 0) + (x > 3) + (x > 8)$	От 0 до 15	5

11.2. Целочисленный кубический корень

В случае вычисления кубического корня метод Ньютона работает существенно хуже в силу более сложной итеративной формулы

$$x_{n+1} = \frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right),$$

а кроме того, как обычно, возникает проблема поиска хорошего начального приближения x_0.

Однако существует аппаратный алгоритм, приведенный в листинге 11.5, который также легко реализуется программно.

Листинг 11.5. Аппаратный алгоритм поиска целочисленного кубического корня

```
int icbrt(unsigned x)
{
    int s;
    unsigned y, b;
    s = 30;
    y = 0;
    while (s >= 0) {  // 11 итераций
        y - 2*y;
        b = (3*y*(y + 1) + 1) << s;
        s = s - 3;
        if (x >= b) {
            x = x - b;
            y = y + 1;
        }
    }
    return y;
}
```

Три команды *сложения с 1* можно заменить командами *или* с 1, поскольку увеличиваемые значения четны. Однако даже при таких изменениях остается сомнительной его применимость для аппаратной реализации, в основном из-за наличия умножения $y*(y+1)$.

Этого умножения легко избежать путем определенной оптимизации. Введем еще одну беззнаковую переменную y2, которая будет содержать значение y^2 и обновляться при каждом изменении y. Непосредственно перед присвоением y = o вставим присвоение y2 = o, а перед y = 2*y — присвоение y2 = 4*y2. Присвоение значения переменной b заменим выражением b = (3* (y2+y) +1)<<s, а непосредственно перед y = y+1 вставим y2 = y2+2*y+1. Полученная в результате программа не содержит умножений, за исключением умножения на малую константу, которое можно заменить *сдвигом* и *сложением*. Эта программа содержит три *сложения* с 1, которые можно заменить командой *или*. Такая программа будет работать быстрее, если только умножение в вашем компьютере не выполняется за два такта (или быстрее).

Будьте осторожны: в [19] указывается, что этот код неработоспособен при непосредственной адаптации его для 64-битовых машин. Присвоение b в этом случае может вызвать переполнение. Этой проблемы можно избежать, убрав сдвиг на s позиций в присвоении b, вставив после присвоения b выражение bs = b<<s и заменив две строки if (x>=b) {x=x-b... строками if (x>=bs&&b==(bs>>s)) {x=x-bs....

11.3. Целочисленное возведение в степень

Вычисление x^n бинарным разложением n

Хорошо известен метод вычисления x^n, где n — неотрицательное целое число, использующий двоичное представление числа п. Этот метод применим к вычислению выражений вида $x \bullet x \bullet x \bullet \ldots \bullet x$, где • представляет собой любой ассоциативный оператор, такой, как сложение, умножение (включая умножение матриц) или конкатенация строк (с использованием записи ('ab')^3='ababab'). В качестве примера рассмотрим вычисление $y = x^{13}$. Поскольку 13 в двоичном представлении имеет вид 1 101 (т.е. 13 равно 8+4+1), то

$$x^{13} = x^{8+4+1} = x^8 \cdot x^4 \cdot x^1.$$

Таким образом, x^{13} можно вычислить, как показано ниже.

$$\begin{aligned} t_1 &\leftarrow x^2 \\ t_2 &\leftarrow t_1^2 \\ t_3 &\leftarrow t_2^2 \\ y &\leftarrow t_3 \cdot t_2 \cdot x \end{aligned}$$

Для этого потребуется пять умножений, что значительно меньше двенадцати, необходимых при непосредственном перемножении значений x.

Если показатель степени — неотрицательная целая переменная, данный метод можно реализовать в виде функции, показанной в листинге 11.6.

Листинг 11.6. Вычисление x^n бинарным разложением n

```
int iexp(int x, unsigned n)
{
    int p, y;
    y = 1; // Инициализация результата
    p = x; // и переменной p
    while (1) {
        if (n & 1)
            y = p*y;      // Если n нечетно, множим на p
        n = n >> 1;       // Позиция очередного бита п
        if (n == 0)
            return y;     // Если больше нет битов п
        p = p*p;          // Степень для очередного бита п
    }
}
```

Количество умножений, которые надо выполнить при использовании данного метода при показателе степени $n \ge 1$, равно

$$\lfloor \log_2 n \rfloor + \text{nbits}(n) - 1.$$

Это не всегда минимально необходимое число умножений. Например, при $n = 27$ метод бинарного разложения приводит к вычислению

$$x^{16} \cdot x^8 \cdot x^2 \cdot x^1,$$

что требует выполнения семи умножений. Однако если воспользоваться схемой

$$\left(\left(x^3\right)^3\right)^3,$$

то будет достаточно шести умножений. Наименьшее значение показателя степени, при котором метод бинарного разложения оказывается неоптимальным, — 15 (подсказка: $x^{15} = (x^3)^5$).

Вероятно, это покажется неожиданным, но простой метод поиска оптимальной последовательности умножений для вычисления У для произвольного n неизвестен. Единственный известный сегодня метод включает обширный поиск. Данная проблема подробно рассматривается в [39, раздел 4.6.3].

Метод бинарного разложения имеет версию, в которой бинарное представление степени сканируется слева направо [53, 32], что аналогично методу преобразования двоичного представления в десятичное слева направо. Инициализируем результат y значением 1 и сканируем двоичное представление показателя степени слева направо. Когда нам встречается нулевой бит, возводим y в квадрат; единичный бит приводит к возведению y в квадрат и умножению на x. Таким образом, $x^{13} = x^{1101_2}$ вычисляется так:

$$\left(\left(\left(1^2 \cdot x\right)^2 \cdot x\right)^2\right)^2 \cdot x .$$

Этот метод требует того же количества умножений, что и метод сканирования справа налево из листинга 11.6.

2" в Fortran

Компилятор IBM XL Fortran использует следующее определение функции:

$$\text{pow2}(n) = \begin{cases} 2^n, & 0 \le n \le 30, \\ -2^{31}, & n = 31, \\ 0, & n < 0 \text{ или } n \ge 32. \end{cases}$$

Здесь предполагается, что и и, и результат возведения в степень рассматриваются как знаковые целые числа. Стандарт ANSI/ISOFortran требует, чтобы результат был нулевым при $n<0$. Данное выше определение при $n > 31$ представляется разумным в том плане, что это корректный результат вычислений по модулю 2^{32}, и согласуется с тем, что дают многократные умножения.

Стандартный способ вычисления 2^n состоит в размещении числа 1 в регистре и сдвиге влево на n позиций. Этот способ не удовлетворяет определению Fortran, поскольку обычно величина сдвига рассматривается по модулю 64 или модулю 32 (на 32-битовом компьютере), что приводит к неверному результату для больших или отрицательных значений смещения.

Если ваш компьютер оснащен командой подсчета *количества ведущих нулевых битов,* то вычисление $\text{pow2}(n)$ может быть выполнено следующим образом:

$x \leftarrow \text{nlz}\left(n \overset{u}{\gg} 5\right)$; // $x \leftarrow 32$ при n из диапазона от 0 до 31; в противном случае $x < 32$

$x \leftarrow x \overset{u}{\gg} 5$; // $x \leftarrow 1$ при n из диапазона от 0 до 31; в противном случае $x = 0$

$pow2 \leftarrow x \ll n$;

В данном случае операция *сдвига вправо* — беззнаковая, даже если n является знаковой величиной.

Если компьютер не имеет команды nlz, вместо нее можно использовать предикат $x = 0$ (см. раздел 2.11), заменив выражение $x \overset{u}{\gg} 5$ выражением $x \overset{u}{\gg} 31$. Возможно, луч-

шим методом реализации предиката $0 < x < 31$ является использование его эквивалентности предикату $x \overset{и}{<} 32$ с упрощением выражения для него из раздела 2.11 до $\neg x \& (x - 32)$. Такой метод дает нам решение из пяти команд (четырех, если машина оснащена командой *и-не*).

$$x \leftarrow \neg n \& (n - 32); \; // x < 0 \text{ тогда и только тогда, когда } 0 \le n \le 31$$

$$x \leftarrow x \overset{и}{\gg} 31; \quad // x = 1 \text{ при } 0 \le n \le 31, \text{ иначе } 0$$

$$pow2 \leftarrow x \ll n;$$

11.4. Целочисленный логарифм

Под "целочисленным логарифмом" подразумевается функция $\lfloor \log_b x \rfloor$, где x — положительное целое число, а b — целое число, не меньшее 2. Обычно $b=2$ или 10; дадим таким функциям имена ilog2 и ilog10 соответственно.

Вполне уместно расширить определение функции для $x=0$, полагая $\text{ilog}(0) = -1$ [7]. Для такого определения имеется ряд причин.

- Функция $\text{ilog2}(x)$ оказывается при этом тесно связанной с функцией *количества ведущих нулевых битов* $\text{nlz}(x)$, так, что если одна из этих функций реализована аппаратно, то вычисление другой — задача совсем несложная: $\text{ilog2}(x) + \text{nlz}(x) = 31$.
- При этом легко вычисляется значение $\lceil \log(x) \rceil$, если воспользоваться формулой $\lceil \log(x) \rceil = \text{ilog}(x-1) + 1$, из которой вытекает, что $\text{ilog}(0) = -1$.
- Такое определение делает справедливым для $x=1$ (но не для $x=0$) тождество $\text{ilog2}(x \div 2) = \text{ilog2}(x) - 1$.
- Это определение сохраняет справедливость тождества $\lfloor \log_{10} x \rfloor = \lfloor (\log_{10} 2) \log_2 x \rfloor$.
- При таком определении возможные значения функции $\text{ilog}(x)$ представляют собой небольшое компактное множество (от -1 до 31 для функции $\text{ilog2}(x)$ на 32-разрядном компьютере при беззнаковом x), что позволяет использовать их для индексации таблиц.
- Это определение естественным путем вытекает из ряда алгоритмов вычисления $\text{ilog2}(x)$ и $\text{ilog10}(x)$.

К сожалению, данное определение некорректно для определения "количества цифр числа x", которое равно $\text{ilog}(x)+1$ для всех x, кроме 0. Однако в силу большого количества преимуществ данное определение для $x=0$ можно считать исключением из правила.

Для отрицательных x функция $\text{ilog}(x)$ не определена. Для расширения области определения данная функция рассматривается как отображающая беззнаковые значения на знаковые, а в этом случае отрицательный аргумент невозможен.

Целочисленный логарифм по основанию 2

Вычисление $\text{ilog2}(x)$, по сути, то же, что и вычисление количества ведущих нулевых битов, которое рассматривалось в разделе *5.3.* Все алгоритмы из этого раздела могут быть легко преобразованы для вычисления $\text{ilog2}(x)$ путем вычисления $\text{nlz}(x)$ с последующим вычитанием этого значения из 31 (в случае алгоритма из листинга 5.10 достаточно заменить строку `return pop(~x)` строкой `return pop(x)-1`.

Целочисленный логарифм по основанию 10

Эта функция применяется при преобразовании числа для включения его в строку с удаленными начальными нулями. Этот процесс состоит в последовательном делении на 10, что дает старшую десятичную цифру. Однако лучше заранее знать, из скольких цифр состоит число, с тем, чтобы избежать размещения полученной строки во временной области для подсчета ее длины с последующим перемещением в нужное место.

Для вычисления $\text{ilog10}(x)$ вполне применим метод поиска в таблице. Вполне возможно использование бинарного поиска, но таблица так мала, что, пожалуй, лучше воспользоваться последовательным поиском. Такой метод использован в приведенном в листинге 11.7 алгоритме.

Листинг 11.7. Целочисленный логарифм по основанию 10, метод поиска в таблице

```
int ilog10(unsigned x)
{
    int i;
    static unsigned table[11] =
        {0, 9, 99, 999, 9999,
         99999, 999999, 9999999, 99999999, 999999999,
         OxFFPFFFFF
        };
    for (i = -1; ; i++) {
        if (x <= table[i+1]) return i;
    }
}
```

На компьютере с базовым набором RISC-команд эта программа требует выполнения $9 + 4\lfloor \log_{10} x \rfloor$ команд, так что максимальное количество выполняемых команд — 45.

Программа из листинга 11.7 может быть легко преобразована в версию без использования таблицы. Выполнимая часть такой программы представлена в листинге 11.8. Этот метод хорошо работает на компьютере с быстрым умножением на 10.

Листинг 11.8. Целочисленный логарифм по основанию 10, метод умножений на 10

```
P = 1;
for (i = -1; i <= 8; i++) {
    if (x < p) return i;
    p = 10*p;
}
return i;
```

Такая программа требует выполнения примерно $10 + 6\lfloor \log_{10} x \rfloor$ команд на компьютере с базовым RISC-набором (считая *умножение* одной командой). Для x из диапазона от 10 до 99 требуется выполнение 16 команд.

Можно применить бинарный поиск, что позволит избавиться от циклов и не использовать таблицу. Такой алгоритм может сравнивать x с 10^4, затем с 10^2 или с 10^6 и т.д., до тех пор пока не будет найден показатель степени n, такой, что $10^n < x < 10^{n+1}$. В этом случае нам потребуется выполнение от 10 до 18 команд, четыре или пять из которых будут командами ветвления (с учетом последнего безусловного перехода).

Программа, показанная в листинге 11.9, представляет собой модификацию бинарного поиска, которая имеет максимум четыре ветвления на каждом из путей и написана в расчете на работу в первую очередь с небольшими значениями x. Она требует выполнения шести базовых RISC-команд для $10 \le x < 99$ и от 11 до 16 команд при $x \ge 100$.

Листинг 11.9. Целочисленный логарифм по основанию 10, модифицированный бинарный поиск

```
int ilog10(unsigned x)
{
    if (x > 99)
        if (x < 1000000)
            if (x < 10000)
                return 3 + ((int)(x - 1000) >> 31);
            else
                return 5 + ((int)(x - 100000) >> 31);
        else
            if (x < 100000000)
                return 7 + ((int)(x - 10000000) >> 31);
            else
                return 9 + ((int)((x-1000000000)&~x) >> 31);
    else
        if (x > 9) return 1;
        else return ((int)(x - 1) >> 31);
}
```

Команда *сдвига* в данной программе — знаковая (именно поэтому в программе использовано приведение типа (int)). Если в вашем компьютере эта команда отсутствует, можно воспользоваться одним из описанных далее вариантов, в которых используется беззнаковый сдвиг. Далее приведены три варианта замены первого оператора return. К сожалению, первые два из них требуют наличия команды *вычитания из непосредственно задаваемого числа,* которая на большинстве машин отсутствует. Последний вариант использует сложение с большой константой (две команды), но это не так важно для остальных операторов return, которые в любом случае работают с большими константами. Величина этой константы — $2^{31} - 1000$.

```
return 3 - ((x - 1000) >> 31);
return 2 + ( (999 - x) >> 31);
return 2 + ( (X + 2147482648) >> 31) ;
```

Четвертый оператор return можно заменить следующим:

```
return 8 + ((x + 1147483648) | x) >> 31;
```

Здесь большая константа равна $2^{31} - 10^9$. Такая замена позволяет избежать использования команды *и-не* и знакового сдвига.

Последняя конструкция `if-else` может быть заменена одной из приведенных ниже.

```
return ((int)(x - 1) >> 31) | ((unsigned)(9 - x) >> 31);
return (x > 9) + (x > 0) - 1;
```

Обе они позволяют сэкономить команду ветвления.

Если компьютер оснащен командой вычисления $\text{nlz}(x)$ или $\text{ilog2}(x)$, то для вычисления $\text{ilog10}(x)$ существуют более эффективные и интересные способы. Так, например, программа из листинга 11.10 позволяет сделать это с помощью двух поисков в таблице [7].

Листинг 11.10. Целочисленный логарифм по основанию 10 из логарифма по основанию 2, метод двойного поиска в таблице

```
int ilog10(unsigned x)
{
    int y;
    static unsigned char table1[33] = {
        9, 9, 9, 8, 8, 8,
        7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
        2, 2, 2, 1, 1, 1, 0, 0, 0, 0
    };
    static unsigned table2[10] = {
        1, 10, 100, 1000, 10000,
        100000, 1000000, 10000000, 100000000, 1000000000
    };
    y = table1[nlz(x)];
    if (x < table2[y]) y = y - 1;
    return y;
}
```

Таблица `table1` дает нам приближение $\text{ilog10}(x)$. Это приближение в основном правильное, однако для некоторых значений x (равного 0 и лежащего в диапазонах от 8 до 9, от 64 до 99, от 512 до 9999, от 8192 до 9999 и т.д.) оно завышено на 1. Если требуется коррекция получаемого значения, то используется таблица `table2`.

В этой схеме используется 73 байта для хранения таблиц, а код состоит из шести команд на машине IBM System/370 [7] (чтобы достичь этого, значения в таблице `table1` должны быть в четыре раза больше показанных). На RISC-компьютере с наличием команды *вычисления количества ведущих нулевых битов* требуется выполнение около десяти команд. Другие рассматривающиеся далее методы, по сути, являются вариантами данного.

Первый вариант состоит в том, чтобы избежать применения оператора `if`. При наличии команды *установки бита, если значение меньше беззнакового значения,* избежать ветвления можно и при использовании программы из листинга 11.10, но описываемый далее метод позволяет сделать это и на других компьютерах, не оснащенных столь экзотической командой.

Метод состоит в замене инструкции if вычитанием после сдвига вправо на 31 позицию, что эквивалентно вычитанию знакового бита. Сложности возникают только при больших значениях $x \ge 2^{31} + 10^9$; для решения этих проблем в `table2` внесен дополнительный элемент (см. листинг 11.11).

Листинг 11.11. Целочисленный логарифм по основанию 10 из логарифма по основанию 2, метод двойного поиска в таблице без условных переходов

```
int ilog10(unsigned x)
{
    int y;
    static unsigned char table1[33] = {
        10, 9, 9, 8, 8, B,
        7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
        2, 2, 2, 1, 1, 1, 0, 0, 0, 0
    };
    static unsigned table2[11] = {
        1, 10, 100, 1000, 10000,
        100000, 1000000, 10000000, 100000000, 1000000000,
        0
    };
    y = table1[nlz(x)];
    y = y - ((x - table2[y]) >> 31);
    return y;
}
```

Эта программа требует выполнения примерно 11 команд на RISC-компьютере, оснащенном командой *вычисления количества ведущих нулевых битов,* но в остальном обладающей "базовым" набором команд. Эту программу легко преобразовать так, чтобы для $x=0$ она возвращала не значение -1, а значение 0 (для этого достаточно изменить последний элемент таблицы `table1` на 1, т.е. заменить "`0, 0, 0, 0`" на "`0, 0, 0, 1`").

Еще один вариант состоит в замене первой таблицы вычитанием, умножением и сдвигом. Это оказывается возможным, так как $\log_{10} x$ и $\log_2 x$ связаны соотношением $\log_{10} x = \log_{10} 2 \cdot \log_2 x$, где $\log_{10} 2 \approx 0.30103$. Таким образом, $\text{ilog10}(x)$ можно вычислить путем вычисления $\lfloor c\,\text{ilog2}(x) \rfloor$ с некоторой константой c и последующей коррекции полученного результата с использованием таблицы (подобно тому, как в листинге 11.11 для этого использовалась таблица table2).

Рассмотрим вопрос выбора константы более подробно. Итак, пусть $\log_{10} 2 = c + \varepsilon$, где $c > 0$ — рациональное приближение $\log_{10} 2$, представляющее собой подходящий множитель, и $\varepsilon > 0$. Тогда для $x > 0$

$$
\begin{aligned}
\text{ilog10}(x) &= \lfloor \log_{10} x \rfloor = \lfloor (c+\varepsilon)\log_2 x \rfloor \\
\lfloor c \log_2 x \rfloor &\le \text{ilog10}(x) = \lfloor c \log_2 x + \varepsilon \log_2 x \rfloor \\
\lfloor c\,\text{ilog2}(x) \rfloor &\le \text{ilog10}(x) < \lfloor c(\text{ilog2}(x)+1) + \varepsilon \log_2 x \rfloor < \\
&< \lfloor c\,\text{ilog2}(x) + c + \varepsilon \log_2 x \rfloor < \\
&\le \lfloor c\,\text{ilog2}(x) \rfloor + \lfloor c + \varepsilon \log_2 x \rfloor + 1.
\end{aligned}
$$

Таким образом, если выбрать c так, что $c + \varepsilon \log_2 x < 1$, то $\lfloor c\,\text{ilog2}(x) \rfloor$ аппроксимирует функцию $\text{ilog10}(x)$ с ошибкой, равной 0 или +1. Кроме того, если полагать, что $\text{ilog2}(0) = \text{ilog10}(0) = -1$, то $\lfloor c\,\text{ilog2}(0) \rfloor = \text{ilog10}(0)$, так как $0 < c < 1$, поэтому нет необходимости в каком-то отдельном рассмотрении этого случая (конечно, могут быть и другие определения, которые также не будут требовать каких-либо дополнительных действий, например $\text{ilog2}(0) = \text{ilog10}(0) = 0$).

Так как $e = \log_{10} 2 - c$, необходимо выбрать c таким, что

$$c + (\log_{10} 2 - c)\log_2 x < 1 \text{ или}$$
$$c(\log_2 x - 1) > (\log_{10} 2)\log_2 x - 1 .$$

Это условие выполняется при $x = 1$ (так как $c < 1$) и 2. Для больших значений x требуется

$$c > \frac{(\log_{10} 2)\log_2 x - 1}{\log_2 x - 1}.$$

Наиболее строгое условие накладывается на c при больших x. На 32-битовом компьютере $x < 2^{32}$, так что c должно удовлетворять условию

$$c > \frac{0.30103 \cdot 32 - 1}{32 - 1} \approx 0.27848 .$$

Так как $e > 0$, $c < 0.30103$, из соображений удобства можно выбрать значение $c = 9/32 = 0.28125$. Эксперименты показывают, что значения 5/16 и 1/4 действительно не подходят для нашего приближения c.

Это приводит нас к схеме, показанной в листинге 11.12, в которой используется заниженная оценка, корректируемая прибавлением 1. Она требует выполнения примерно 11 RISC-команд, среди которых команда *вычисления количества ведущих нулевых битов (умножение* считается одной командой).

Листинг 11.12. Целочисленный логарифм по основанию 10 из логарифма по основанию 2, метод поиска в таблице

```
static unsigned table2 [10] = {
    O, 9, 99, 999, 9999,
    99999, 999999, 9999999, 99999999, 999999999
};
y = (9* (31 - nlz(x))) >> 5;
if (x > table2[y+1]) y = y + 1;
return y;
```

В этой программе можно избежать условных переходов, но при этом вновь возникнут сложности при больших значениях $x > 2^{31} + 10^9$, справиться с которыми можно различными способами. Один из способов состоит в использовании другого множителя, а именно $19/64$, и несколько измененной таблицы. Соответствующая программа показана в листинге 11.13. Она требует вышолнения примерно 11 RISC-команд, среди которых команда *вычисления количества ведущих нулевых битов (умножение* считается одной командой).

Листинг 11.13. Целочисленный логарифм по основанию 10 из логарифма по основанию 2, метод поиска в таблице, без команд ветвления

```
int ilog10 (unsigned x)
{
    int y;
    static unsigned table2 [11] = {
        0, 9, 99, 999, 9999,
        99999, 999999, 9999999, 99999999, 999999999,
        OxFFFFFFFF
    };
    y = (19*(31 - nlz(x))) >> 6;
```

```
    y = y + ((table2[y+1] - x) >> 31);
    return y;
}
```

Другой способ состоит в использовании команды *или с х к* разностью в качестве операндов для обеспечения установки знакового бита при $x \geq 2^{31}$. Таким образом, при использовании этого способа вторая выполнимая строка в листинге 11.12 должна быть заменена следующей:

```
y = y + (((table2[y+1] - x) | x) >> 31);
```

Этот способ предпочтительнее, если умножение на 19 оказывается существенно сложнее умножения на 9 (реализуемые в виде последовательностей *сдвигов* и *сложений).*

На 64-разрядном компьютере следует выбрать *с,* удовлетворяющее условию

$$c > \frac{0.30103 \cdot 64 - 1}{64\text{-}1} \approx 0.28993.$$

Таким значением может быть $19/64 = 0.296875$ и, как показывают эксперименты, более грубого приближения не существует. В результате будет получена следующая программа:

```
unsigned table2[20] = {0, 9, 99, 999, 9999, ...,
    9999999999999999999};
y = ((19*(63 - nlz(x)) >> б;
y = y + ((table2[y+1] - x) >> 63;
return y;
```

ГЛАВА 12

СИСТЕМЫ СЧИСЛЕНИЯ С НЕОБЫЧНЫМИ ОСНОВАНИЯМИ

В этой главе рассматривается несколько необычных позиционных систем счисления. В основном такие системы представляют собой не более чем интересный курьез, не имеющий практического применения. Наше рассмотрение ограничивается целыми числами, однако его легко распространить и на цифры после точки, что обычно (хотя и не всегда) обозначает нецелые числа.

12.1. Основание -2

Использование системы счисления по основанию -2 дает возможность выражать как положительные, так и отрицательные числа без явного указания их знака или, например, указания отрицательного веса старшего значащего бита [40]. При такой записи используются цифры 0 и 1, как и в случае системы счисления по основанию 2. Таким образом, значение, представленное строкой из нулей и единиц, трактуется как

$$(a_n \ldots a_3 a_2 a_1 a_0) = a_n(-2)^n + \ldots + a_3(-2)^3 + a_2(-2)^2 + a_1(-2)^1 + a_0 .$$

Из этого определения видно, что процедура поиска представления числа в системе счисления по основанию -2 должна выполнять последовательное деление на -2, записывая получаемые остатки. Используемое деление должно быть таким, которое всегда дает в остатке 0 или 1 (используемые в записи чисел цифры), т.е. это модульное деление. В качестве примера покажем, как найти представление числа -3 по основанию -2.

$$\frac{-3}{-2} = 2 \text{ rem } 1$$
$$\frac{2}{-2} = -1 \text{ rem } 0$$
$$\frac{-1}{-2} = 1 \text{ rem } 1$$
$$\frac{1}{-2} = 0 \text{ rem } 1$$

Поскольку мы достигли нулевого частного, процесс на этом завершается (при его продолжении все остальные частные и остатки будут равны 0). Таким образом, считывая значения остатков снизу вверх, получаем, что число -3, записанное в системе счисления с основанием -2, равно 1101.

В табл. 12.1 слева показаны десятичные числа, соответствующие каждой из битовых последовательностей от 0000 до 1111 в системе счисления с основанием -2, а справа — представление в этой системе счисления десятичных чисел в диапазоне от -15 до +15.

Таблица 12.1. Преобразование десятичных чисел в числа про основанию -2

п (основание -2)	п (десятичное)	п (десятичное)	п (основание -2)	п (основание -2)
0	0	0	0	0
1	1	1	1	11
10	-2	2	ПО	10
11	-1	3	111	1101
100	4	4	100	1100
101	5	5	101	1111
НО	2	6	11010	1110
111	3	7	11011	1001
1000	-8	8	11000	1000
1001	-7	9	11001	1011
1010	-10	10	11110	1010
1011	-9	11	11111	110101
1100	-4	12	11100	110100
1101	-3	13	11101	110111
1110	-6	14	10010	110110
1111	-5	15	10011	110001

Не так очевидно, что 2" возможных битовых строк длиной *n* однозначно представляют все целые числа из некоторого диапазона, но это свойство может быть доказано по индукции. Индуктивная гипотеза в данном случае состоит в том, что *n*-битовые слова представляют все целые числа из диапазона

$$\text{от } -\left(2^{n+1}-2\right)/3 \text{ до } \left(2^{n}-1\right)/3 \text{ для четных и,} \quad (1,\text{а})$$

$$\text{от } -\left(2^{n}-2\right)/3 \text{ до } \left(2^{n+1}-1\right)/3 \text{ для нечетных п.} \quad (1,\text{б})$$

Предположим сначала, что *n* четно. Для $n=2$ представляемыми числами являются 10, 11, 00 и 01 по основанию -2, т.е. десятичные числа -2, −1, 0, 1. Это согласуется с формулой (1,а), причем каждое из чисел диапазона представлено только один раз.

Слово из $n+1$ битов при ведущем бите 0 может представлять все числа, задаваемые формулой (1,а). Если ведущий бит равен 1, то это слово может представлять все эти же целые числа, смещенные на (-2)" = 2^n . Новый диапазон равен

$$\text{от } 2^n-\left(2^{n+1}-2\right)/3 \text{ до } 2^n+\left(2^n-1\right)/3$$

или

$$\text{от } \left(2^{n}-1\right)/3+1 \text{ до } \left(2^{n+2}-1\right)/3.$$

Этот диапазон является смежным с диапазоном (1,а), так что слово размером и +1 битов однозначно представляет все целые числа из диапазона

$$\text{от } -\left(2^{n+1}-2\right)/3 \text{ до } \left(2^{n+2}-1\right)/3.$$

Это согласуется с (1,б), если заменить *n* на $n+1$.

Доказательство того, что (1,а) следует из (1,б) при нечетном *n* и что все целые числа диапазона представлены однозначно, проводится аналогично.

В случае сложения и вычитания используются обычные правила: 0 + 1 = 1 и 1-1 = 0. Поскольку 2 записывается как 110, а -1 — как 11, применимы дополнительные правила, которых наряду с обычными вполне достаточно для проведения арифметических вычислений.

```
 1+1=110
11+1=0
1+1+1=111
 0-1=11
11-1=10
```

При сложении и вычитании иногда имеется два бита переноса. Биты переноса добавляются к столбцу даже при вычитании. Их удобно располагать над следующим битом слева и использовать (по возможности) упрощение 11+1=0. Если 11 переносится в столбец, который содержит два 0, записываем в нем 1 и переносим 1 дальше. Приведем конкретные примеры сложения и вычитания.

```
        Сложение                         Вычитание

11  1 11   11                   1 11  1    1
    1 0 1 1 1       1 9             1 0 1 0 1     2 1
+ 1  1  0  1  0  1 +(-11)      - 1  O  1  1  1  0 -(-38)
  --------------- ------         ---------------- -----
 0  1  1  0  0  0     8         1  0  0  1  1  1  1     5 9
```

Возможны только следующие значения битов переносов: О, 1 и 11. Переполнение происходит тогда, когда имеется перенос (1 или 11) из старшей позиции. Это замечание справедливо как для сложения, так и для вычитания.

Поскольку возможны три варианта переноса, сумматор по основанию счисления −2 существенно сложнее сумматора в дополнительном коде.

Существует два способа изменить знак числа. Его можно прибавить к самому себе, сдвинутому влево на одну позицию (т.е. умноженному на -1), либо вычесть его из 0. В этом случае нет такого простого и удобного правила, как "дополнить и прибавить 1", присущего арифметике на основе дополнительного кода. При работе с дополнительным кодом это правило используется для создания вычитающего модуля на основе сумматора (для вычисления $A - B$ формируется сумма $A + \bar{B} + 1$). В случае работы с числами в системе счисления по основанию −2 построить такое простое устройство невозможно.

Умножение в системе счисления по основанию -2 достаточно простое. При этом используются простые правила: 1x1 = 1 и умножение на 0 дает 0; сложение в столбик выполняется с применением правил сложения чисел в системе счисления по основанию -2.

Деление, однако, оказывается существенно сложнее. Это действительно очень интересная и сложная задача — разработка реального аппаратного алгоритма деления, т.е. алгоритма, основанного на повторяющихся вычитаниях и сдвигах. В листинге 12.1 показан алгоритм, предназначенный для работы на 8-битовом компьютере. Он реализует модульное деление (когда остаток от деления неотрицателен).

Листинг 12.1. Деление в системе счисления с основанием -2

```
int divbm2(int n, int d)      // q = n/d по основанию -2
{
   int r, dw, c, q, i;
   r = n;                     // Инициализация остатка
   dw = (-128)*d;             // Позиция d
   c = (-43)*d;               // Инициализация компаранда
```

```
    if (d > 0) c = c + d;
    q = 0;                    // Инициализация частного
    for (i = 7; i >= 0; i--) {
        if (d > 0 ^ (i&1)== 0 ^ r >= c) {
            q = q | (1 << i);// Установка бита частного
            r = r - dw;       // Вычитание сдвинутого d
        }
        dw = dw/(-2);             // Позиция d
        if (d > 0) c = c - 2*d; // Установка компаранда
        else c = c + d;           // для следующей итерации
        c = c/(-2);
    }
    return q;                 // Возврат частного
                              // по основанию -2
                              // Остаток r, 0 <= r < |d|
}
```

Хотя эта программа написана на С и протестирована на машине с дополнительным кодом, это не играет никакой роли — данный код следует рассматривать абстрактно. Входные величины n и d и все внутренние переменные, за исключением q, являются просто числами без какого-либо конкретного представления. Выходная величина q является строкой битов, интерпретируемой как число в системе счисления по основанию -2.

Здесь требуется небольшое пояснение. Если бы входные величины были числами в системе счисления по основанию -2, то алгоритм было бы очень трудно выразить в выполнимом виде. Например, проверка "`if (d>0)`" должна была бы установить, находится ли старший значащий бит числа d в четной позиции. Сложение в выражении "`c = c+d`" также должно было быть реализовано по правилам сложения в системе счисления по основанию -2. Такой код был бы слишком сложен для чтения. Поэтому, рассматривая данный алгоритм, вы должны воспринимать n и d как числа без конкретного представления. Приведенный код просто показывает, какие именно арифметические операции должны быть выполнены, независимо от используемого способа кодирования. Если бы числа были представлены в системе счисления по основанию -2, как при аппаратной реализации данного алгоритма, умножение на -128 представляло бы собой сдвиг влево на семь позиций, а деление на -2 — сдвиг вправо на одну позицию.

В качестве примеров приведем вычисляемые этим кодом значения:

$\text{divbm2}(6,2) = 7$ (шесть, деленное на 2, равно 111_{-2})

$\text{divbm2}(-4,3) = 2$ (минус четыре, деленное на 3, равно 10_{-2})

$\text{divbm2}(-4,-3) = 6$ (минус четыре, деленное на минус три, равно 110_{-2})

Шаг `q = q|(1<<i);` представляет собой просто установку i-го бита числа q. Следующая строка — `r = r-dw;` — представляет уменьшение остатка на сдвинутое влево значение делителя d.

Этот алгоритм трудно описать детально, но постараемся представить главную идею.

Рассмотрим определение значения первого бита частного, бита 7 числа q. При основании системы счисления, равном -2, 8-битовое число, у которого старший бит равен 1, находится в диапазоне от -170 до –43. Таким образом, игнорируя возможность переполнения, первый (старший) бит частного будет равен 1 тогда (и только тогда), когда частное алгебраически не превышает –43.

Поскольку $n = qd + r$ и для положительного делителя справедливо соотношение $r < d - 1$, то для положительного делителя первый бит частного будет равен 1 тогда и только тогда, ко-

гда $n \leq -43d + (d-1)$ или и $< -43d + d$. Для отрицательного делителя первый бит частного будет равен 1 тогда и только тогда, когда $n > -43d$ (при модульном делении $r > 0$).

Таким образом, первый бит частного равен 1 тогда и только тогда, когда

$$\left(d > 0 \,\&\, \neg(n \geq -43d + d)\right) | \left(d < 0 \,\&\, n \geq -43d\right).$$

Игнорируя возможное нулевое значение *d,* можем записать это выражение как

$$d > 0 \oplus n \geq c,$$

где $c = -43d + d$ при положительном *d* и $c = -43d$ при *d* отрицательном.

Так выглядит логика определения бита частного для нечетной позиции. Для четной позиции логика обратная. Из-за этого условие в операторе if включает проверку `(i&1)==0` (напомним, что символ `^` в языке С означает операцию *исключающее или).*

При каждой итерации с устанавливается равным наименьшему (наиболее близкому к 0) целому числу с единичным битом в позиции i после деления на d. Если текущий остаток `r` превышает это значение, бит i числа q устанавливается равным 1, а `r` корректируется путем вычитания числа, представляющего собой 1 в этой позиции, умноженного на делитель d. Реальное умножение при этом не требуется — d просто соответствующим образом позиционируется, после чего выполняется вычитание.

Алгоритм трудно назвать элегантным. Его очень трудно реализовать из-за ряда сложений, вычитаний, сравнений и даже умножения (на константу), которые должны быть выполнены в начале. Надеяться на существование "однородного" алгоритма, т.е. алгоритма, в котором действия одинаковы вне зависимости от знаков аргументов, для системы счисления по основанию -2, по-видимому, не приходится. Причина в том, что деление представляет собой, по сути, неоднородный процесс. Рассмотрим простейший алгоритм на основе сдвигов и вычитаний. Такой алгоритм может вовсе не использовать сдвиги и для положительных аргументов использовать только итеративное вычитание делителя из делимого с подсчетом количества вычитаний до тех пор, пока остаток не окажется меньше делителя. Однако, если делимое отрицательно (а делитель положителен), процесс будет состоять в итеративном прибавлении делителя к делимому, пока остаток не станет неотрицательным (частное при этом будет представлять собой количество сложений с обратным знаком). Процесс будет иным, если делитель отрицателен.

Несмотря на это, деление является однородным процессом для представления чисел в прямом коде со знаком. При таком представлении значения чисел положительны, так что алгоритм может просто осуществлять вычитания до тех пор, пока остаток не станет отрицательным, а затем установить бит знака частного равным *исключающему или* знаковых битов делимого и делителя, а знаковый бит остатка — равным знаку делимого (что дает нам обычное деление с отсечением).

Алгоритм, приведенный выше, можно сделать в определенном смысле более однородным, дополняя делитель, если он отрицателен, а затем выполняя упрощенные для случая $d > 0$ шаги. В конце алгоритма выполняется окончательная коррекция. Для модульного деления коррекция изменяет знак частного и оставляет неизменным остаток. Такое изменение алгоритма выносит некоторые проверки за пределы цикла, но особой красоты алгоритму в целом не прибавляет.

Интересно сравнить распространенные представления чисел и числа в системе счисления по основанию -2 в аспекте однородности при выполнении четырех арифметических

операций. Точного определения понятия "однородность" нет, но можно понимать под этим наличие или отсутствие операций, выполняемых в зависимости от знака аргументов. Мы считаем операцию однородной, если знаковый бит результата равен *исключающему или* от знаковых битов аргументов операции. В табл. 12.2 показано, какие из операций и при работе с какими представлениями чисел рассматривают свои аргументы однородно.

Таблица 12.2. Однородность операций при использовании разных представлений чисел

	Знаковые величины	Дополнение до 1	Дополнение до 2	По основанию -2
Сложение	Нет	Да	Да	Да
Вычитание	Нет	Да	Да	Да
Умножение	Да	Нет	Нет	Да
Деление	Да	Нет	Нет	Нет

Сложение и вычитание чисел в дополнительном к 1 представлении выполняется однородно при помощи приема с "переносом в конец слова". В случае сложения все биты, включая знаковый, суммируются по обычным правилам двоичной арифметики, и перенос из самого левого (знакового) бита добавляется к младшему биту результата. Этот процесс всегда корректно завершается (т.е. такое добавление переноса не может привести к новому переносу в позиции знакового бита).

В случае умножения чисел в дополнительном к 2 представлении запись "Да" справедлива только в случае, если нас интересует только правая половина двойного слова произведения.

Завершим это рассмотрение системы счисления по основанию -2 некоторыми наблюдениями, касающимися выполнения преобразований между числами в этой системе счисления и обычными двоичными числами.

Для того чтобы преобразовать число из системы счисления по основанию -2 в двоичное число, формируем слово, состоящее только из битов с положительными весами, и вычитаем из него слово, состоящее из битов с отрицательными весами, используя при этом правила вычитания для двоичной арифметики. Другой метод, который может показаться немного проще, состоит в выделении битов, находящихся в позициях с отрицательными весами, сдвиге их на одну позицию влево и вычитании выделенного числа из исходного с использованием обычных правил вычитания двоичной арифметики.

Для преобразования двоичного числа в число в системе счисления по основанию -2 нужно выделить биты из нечетных позиций (позиций с весом 2^n, где n нечетно), сдвинуть их на одну позицию влево и сложить два числа с использованием правил сложения чисел в системе счисления по основанию -2. Приведем два примера преобразования чисел.

```
     Двоичное из системы           В систему счисления по
   счисления по основанию -2     по основанию -2 из двоичного
        110111 (-13)                  110111 (55)
      - 1 0 1   (двоичное           + 1 0 1   (сложение по
      ───────── вычитание)          ───────── основанию -2)
   ...111110011 (-13)                1001011 (55)
```

В компьютере, с его фиксированным размером слова, эти преобразования работают и для отрицательных чисел, если просто игнорировать перенос из старшей позиции. Для иллюстрации этого можно рассмотреть приведенный выше справа пример как преобразование двоичного числа -9 (размер слова — 6 бит) в систему счисления по основанию -2.

Приведенный выше алгоритм преобразования в число в системе счисления по основанию -2 не поддается легкой программной реализации на двоичном компьютере, поскольку требует выполнения сложения в соответствии с правилами сложения чисел в системе счисления по основанию -2. Шреппель (Schroeppel) [25, item 128] преодолел это препятствие, разработав более ясный и практичный метод выполнения преобразований в обоих направлениях. Для преобразования в двоичное число его метод выглядит следующим образом:

$$B \leftarrow (N \oplus \mathbf{0b10...1010}) - \mathbf{0b10...1010}.$$

Чтобы убедиться, что этот метод работает, рассмотрим число, которое в системе счисления по основанию -2 имеет вид $abcd$. Тогда, если (некорректно) рассматривать его как обычное двоичное число, его значение равно $8a + 4b + 2c + d$. После выполнения операции *исключающее или* это число, рассматриваемое как двоичное, равно $8(1-a) + 4b + 2(1-c) + d$. После (двоичного) вычитания получаем значение $-8a + 4b - 2c + d$, которое представляет собой значение числа $abcd$ в системе счисления по основанию -2.

Формула Шреппеля может быть легко разрешена для выполнения обратного преобразования, давая нам метод преобразования в обратном направлении, требующий выполнения трех команд. Ниже приведены формулы для преобразования в двоичное число на 32-битовой машине.

$$B \leftarrow (N \,\&\, \mathbf{0x55555555}) - (N \,\&\, \neg\mathbf{0x55555555}),$$
$$B \leftarrow N - ((N \,\&\, \mathbf{0xAAAAAAAA}) \ll 1),$$
$$B \leftarrow (N \oplus \mathbf{0xAAAAAAAA}) - \mathbf{0xAAAAAAAA}.$$

Для преобразования числа в систему счисления по основанию -2 используется следующая формула:

$$N \leftarrow (B + \mathbf{0xAAAAAAAA}) \oplus \mathbf{0xAAAAAAAA}.$$

12.2. Основание -1+i

Используя в качестве основания системы счисления $-1+i$, где i обозначает $\sqrt{-1}$, все *комплексные* целые числа (т.е. комплексные числа, действительная и мнимая части которых целые) можно выразить в виде одного "числа" без явного использования знака или других вспомогательных средств. Приятной неожиданностью оказывается то, что это можно осуществить только при помощи цифр 0 и 1, причем все целые числа имеют при этом однозначное представление. Не будем здесь доказывать этот факт, как и многие другие, только опишем вкратце свойства данной системы счисления.

Не так уж тривиально оказывается выяснение того, как следует записать целое число 2 в этой системе счисления[1]. Однако эта задача вполне разрешима путем последовательного деления 2 на основание системы счисления и записи остатков. Но что такое остаток от деления в данном контексте? Мы хотим, чтобы остаток после деления на $-1+i$ был

[1] Попробуйте сделать это самостоятельно, отложив на время книгу.

равен 0 или 1, если это возможно (так, чтобы цифрами были только 0 и 1). Для того чтобы увидеть, что это всегда возможно, предположим, что выполняется деление произвольного комплексного целого числа $a+bi$ на $-1+i$. Требуется найти q и r такие, что q — комплексное целое число, r равно 0 или 1 и

$$a+bi=(q_r+q_i i)(-1+i)+r,$$

где q_r и q_i означают действительную и мнимую части q соответственно. Приравнивая действительные и мнимые части и решая одновременно полученные уравнения, получим

$$q_r=\frac{b-a+r}{2},$$
$$q_i=\frac{-a-Ь+г}{£}.$$

Ясно, что если a и Ъ оба четны или оба нечетны, то выбор $r=0$ дает нам комплексное целое число q. Если же одно из чисел a и Ъ четно, а второе — нет, то комплексное целое число q можно получить при выборе $r=1$.

Таким образом, целое число 2 может быть преобразовано к основанию $-1+i$ так, как показано ниже.

Поскольку действительная и мнимая части целого числа 2 четны, просто выполняем деление, зная, что остаток равен 0:

$$\frac{2}{-1+i}=\frac{2(-1-i)}{(-1+i)(-1-i)}=-1-i \text{ rem } 0.$$

Действительная и мнимая части числа $-1-i$ нечетны, так что остаток опять равен 0:

$$\frac{-1-i}{-1+i} \quad \frac{(-1-i)(-1-i)}{(-1+i)(-1-i)} \quad i \text{ rem } 0.$$

Поскольку теперь действительная и мнимая части i являются соответственно четной и нечетной, остаток будет равен 1. Проще всего учесть его, вычитая 1 из делимого:

$$\frac{i-1}{-1+\imath}=1\,(\text{остаток } 1).$$

Теперь действительная и мнимая части 1 являются соответственно нечетной и четной, так что остаток будет равен 1. Вычитая его из делимого, получаем

$$\frac{1-1}{-1+\imath}=0\,(\text{остаток } 1).$$

Таким образом получено нулевое частное, и процесс на этом завершается. Считывая полученные при делении остатки, получаем, что представление 2 в системе счисления по основанию $-1+i$ имеет вид 1100.

В табл. 12.3 показаны все битовые строки от 0000 до 1111 и их интерпретация в системе счисления по основанию $-1+i$, а также представление в этой системе десятичных чисел от -15 до +15.

Таблица 12.3. Преобразования между десятичной системой счисления и системой счисления по основанию $-1+i$

n (основание -1+0)	п (десятичное)	n (десятичное)	n (основание -1+0)	-n (основание -1+о)
0	0	0	0	0
1	1	1	1	11101
10	$-1+i$	2	1100	11100
11	i	3	1101	10001
100	$-2i$	4	111010000	10000
101	$1-2i$	5	111010001	11001101
ПО	$-1-i$	6	111011100	11001100
111	$-i$	7	111011101	11000001
1000	2+2/	8	111000000	11000000
1001	3+2/	9	111000001	11011101
1010	$1+3i$	10	111001100	11011100
1011	2+3/	11	111001101	11010001
1100	2	12	100010000	11010000
1101	3	13	100010001	1110100001101
1110	$1+i$	14	100011100	1110100001100
1111	$2+i$	15	100011101	1110100000001

Правила сложения в системе счисления по основанию $-1+i$ (кроме тривиальных правил сложения с участием нулевых битов) выглядят следующим образом:

$$
\begin{aligned}
1+1 &= 1100\\
1+1+1 &= 1101\\
1+1+1+1 &= 111010000\\
1+1+1+1+1 &= 111010001\\
1+1+1+1+1+1 &= 111011100\\
1+1+1+1+1+1+1 &= 111011101\\
1+1+1+1+1+1+1+1 &= 111000000
\end{aligned}
$$

При сложении двух чисел наибольшее количество переносов, которое может возникнуть в одном столбце, — шесть, так что максимальная сумма, которая может быть получена в одном столбце, — 8 (111000000). Это делает построение сумматора крайне сложной задачей. Если бы нашелся кто-то разрабатывающий компьютер с комплексной арифметикой, то, вне всяких сомнений, гораздо лучше было бы хранить действительную и мнимую части отдельно[2], причем каждая часть имела бы более разумное представление, например в дополнительном к 2 коде.

12.3. Другие системы счисления

Система счисления по основанию $-1-i$ имеет, по сути, все те же свойства, что и система счисления по основанию $-1+i$, рассмотренная выше. Если некоторая битовая стро-

---

[2] Этот способ был использован в 1940 году в Bell Labs в Complex Number Calculator Джорджа Стибица (George Stibitz) [35].

ГЛАВА 13
КОД ГРЕЯ

13.1. Построение кода Грея

Нельзя ли циклически перебрать все 2" комбинаций из и битов путем изменения при каждой итерации только одного бита? Ответ на этот вопрос — "да"; и именно это свойство определяет коды Грея (Gray codes). Итак, код Грея — это такой способ кодирования целых чисел, при котором закодированное число и число, следующее за ним, отличаются только одним битом. Эта концепция может быть обобщена для любого основания системы счисления, например для десятичных чисел, но в этой главе рассматриваются только бинарные коды Грея.

Хотя существует множество различных вариантов кодов Грея, рассмотрим только один: "двоичный отраженный (рефлексный) код Грея". Именно этот код обычно имеется в виду, когда говорят о неконкретном "коде Грея". Покажем (как обычно, без доказательств), как выполняются основные операции при таком представлении целых чисел, и расскажем о некоторых интересных свойствах кода Грея.

Отраженный двоичный код Грея строится следующим образом. Начинаем со строк О и 1, которые представляют соответственно целые числа 0 и 1.

0
1

Возьмем отражение этих строк относительно горизонтальной оси после приведенного списка и поместим 1 слева от новых записей списка, а слева от уже имевшихся разместим 0.

00
01
11
10

Таким образом получен отраженный код Грея для $n=2$. Чтобы получить код для $n=3$, повторим описанную процедуру и получим

000
001
011
010
ПО
111
101
100

При таком способе построения легко увидеть по индукции по n, что, во-первых, каждая из 2^n комбинаций битов появляется в списке, причем только один раз; во-вторых, при переходе от одного элемента списка к рядом стоящему изменяется только один бит; в-третьих, только один бит изменяется при переходе от последнего элемента списка к первому. Коды Грея, обладающие последним свойством, называются циклическими, и отраженный код Грея обязательно является таковым.

При я > 2 существуют нециклические коды Грея, которые состоят из всех 2^n значений, взятых по одному разу. Примером такого кода может служить следующий: 000 001 011 010 110 100 101 111.

В схеме на рис. 13.1 показаны четырехбитовые числа, закодированные в обычной двоичной системе счисления и кодами Грея. Показаны также формулы для преобразования одного представления в другое на уровне "бит за битом", применимом в аппаратной реализации.

Бинарный *abcd*	Грея *efgh*		
0000	0000		
0001	0001	Код Грея из бинарного	Бинарный из кода Грея
0010	ООП	$e = a$	$a = e$
ООП	0010	$f = a \oplus b$	$b = e®f$
0100	0110	$g = b®c$	$c = e \oplus f \oplus g$
0101	0111	$h = c®d$	$d = e \oplus f \oplus g \oplus h$
0110	0101		
0111	0100		
1000	1100		
1001	1101		
1010	1111		
1011	1110		
1100	1010		
1101	1011		
1110	1001		
1111	1000		

Рис. 13.1. Четырехбитовый код Грея и формулы преобразования

Заметим, что циклический код Грея из n бит остается таковым как после любого циклического сдвига кода по вертикали, так и при любом изменении порядка столбцов. Любая из этих операций дает новый циклический код Грея, отличающийся от остальных, полученных таким же методом. Таким образом, имеется, как минимум, 2" • $n!$ циклических бинарных кодов Грея из n битов (на самом деле их количество превышает данную оценку).

Код Грея и двоичное представление обладают следующим дуальным отношением, очевидным из приведенных на рис. 13.1 формул.

- Бит i числа в коде Грея представляет собой четность бита i и бита слева от него в соответствующем двоичном числе (если слева от бита i нет битов, используем значение 0).
- Бит i двоичного числа представляет собой четность всех битов в позиции i и слева от нее в соответствующем числе кода Грея.

Преобразование двоичного числа в код Грея может быть выполнено при помощи всего двух команд:

$$G \leftarrow B \oplus \left(B \overset{u}{\gg} 1\right).$$

Преобразование кода Грея в двоичное число существенно сложнее:

$$B \leftarrow \bigoplus_{i=0}^{n-1} G \overset{u}{\gg} i .$$

Вы уже встречались с этой формулой в разделе 5.2. Как упоминалось в этом разделе, вычисления по данной формуле можно выполнять так, как показано ниже (для $n=32$).

```
B = G ^ (G >> 1);
B = B ^ (B >> 2);
B = B * (B >> 4);
B = B ^ (B >> 8);
B = B ^ (B >> 16);
```

Таким образом, в общем случае требуется выполнение 2 • [log, л] команд.

Поскольку преобразование двоичных чисел в код Грея выполняется очень просто, генерация последовательных чисел кода Грея представляет собой тривиальную задачу:

```
for (i = 0; i < n; i++) {
    G = i ^ (i >> 1);
    OUtpUt G;
}
```

13.2. Увеличение чисел кода Грея

Логика увеличения 4-битового *двоичного* числа ***abcd*** может быть выражена следующим образом, с использованием обозначений булевой алгебры:

$$d' = \bar{d}$$
$$c' = c \text{®} d$$
$$b' = b \text{®} cd$$
$$a' = a \text{®} bcd$$

Таким образом, один способ построения аппаратного счетчика кода Грея состоит в создании бинарного счетчика с использованием описанной логики и преобразовании его вывода a', b', c' и d' в код Грея при помощи операции *исключающего или* над соседними битами, как показано в схеме на рис. 13.1.

Приведем еще один способ, который может оказаться несколько эффективнее:

$$p = e \oplus f \oplus g \oplus h$$
$$h' = h \oplus \bar{p}$$
$$g' = g \oplus hp$$
$$f' = f \oplus g\bar{h}p$$
$$e' = e \oplus \bar{f}\bar{g}hp$$

Таким образом, в общем случае

$$G'_n = G_n \oplus \left(G_{n-1}\bar{G}_{n-2}\ldots\bar{G}_0 p\right),\ n \geq 2.$$

Поскольку четность p чередуется между 0 и 1, схема счетчика может поддерживать p как отдельный однобитовый регистр и просто инвертировать его при каждой итерации.

Программно наилучший способ поиска последующего элемента G' для целого числа G в коде Грея, вероятно, состоит в конвертации G в двоичное число, его увеличении и обратном преобразовании в код Грея. Еще один интересный и почти такой же по эффективности путь — это определение того, какой бит должен быть изменен в числе G. Вот как выглядит последовательность этих битов, представленная в виде слов, которые при помощи операции *исключающего или* с числом G дают очередное число кода:

1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16

Внимательный читатель распознает в этой последовательности маску, указывающую положение крайнего слева бита, который изменяется при увеличении целого числа 0, 1, 2, 3, ..., соответствующего позиции в приведенной последовательности. Таким образом, при увеличении числа G из кода Грея позиция инвертируемого бита определяется самым левым битом, который изменяется при прибавлении 1 к двоичному числу, соответствующему *G*.

Рассмотренные способы приводят нас к алгоритмам увеличения чисел кода Грея, показанным в листинге 13.1 (оба они сначала преобразуют число *G* в двоичное, что обозначено как index(G)).

Листинг 13.1. Увеличение числа кода Грея

```
B = index(G) ;                          B = index(G);
B = B + 1;                            M = -B & (B + 1) ;
Gp = B ^(B >>1) ;                      Gp = G ^ M;
```

Метод увеличения чисел кода Грея "с карандашом в руке" выглядит следующим образом.

> Начиная справа, найти первый бит, для которого четность данного бита и всех битов слева положительна. Инвертировать бит в данной позиции.

Можно воспользоваться другим, эквивалентным, правилом.

> Пусть *p* — четность слова *G*. Если *p* четно, инвертировать младший бит. Если же *p* нечетно, инвертировать бит слева от крайнего единичного бита справа.

Последнее правило непосредственно выражено приведенными выше булевыми выражениями.

13.3. Отрицательно-двоичный код Грея

Если вы запишете целые числа в системе счисления по основанию -2 и преобразуете их с помощью *сдвига* и *исключающего или* так же, как обычное двоичное число преобразуется в число кода Грея, то полученные числа также дадут код Грея. Трехбитовый код Грея имеет индексы в диапазоне трехбитовых чисел в системе счисления по основанию -2, а именно от -2 до 5; четырехбитовый — от -10 до 5. Полученный таким образом код Грея не является отраженным, но очень близок к нему. Такой четырехбитовый код Грея можно построить, начиная с 0 и 1, отражая их относительно *верха* списка, затем относительно *низа* списка и т.д., чередуя отражения.

Для преобразования чисел такого кода Грея назад в систему счисления по основанию -2 правила, разумеется, те же, что и для преобразования обычного кода Грея в двоичные числа (так как эти операции обратны друг другу, не имеет значения, как именно трактуются битовые строки, участвующие в операциях).

13.4. Краткая история и применение

Коды Грея получили свое название по имени Франка Грея (Frank Gray), физика из Bell Telephone Laboratories, который в 1930-х годах изобрел метод, в настоящее время используемый для передачи цветного телевизионного сигнала, совместимого с существующими методами передачи и получения черно-белого сигнала; т.е. при получении цветного сигнала черно-белым приемником изображение выводится оттенками серого цвета.

Мартин Гарднер (Martin Gardner) [15] рассматривал применение кодов Грея для решения головоломок с китайскими кольцами, ханойскими башнями, и для построения гамильтониановых путей в графе, представляющем гиперкубы. Он также показал, каким образом можно преобразовывать числа из десятичного представления в десятичные коды Грея.

Коды Грея используются в датчиках положения. Представьте себе полосу материала, которая разделена на проводящие и непроводящие области, соответствующие нулям и единицам целых чисел кода Грея. Каждая такая полоса контактирует с проводящей щеткой. Если щетка располагается над разделяющей линией между двумя областями так, что получается неоднозначное считывание позиции, то не имеет никакого значения, как именно будет разрешена данная неоднозначность, поскольку неоднозначным может быть положение только одной щетки (и как бы не определялось ее положение, это будут две соседние области).

Полоса материала может быть серией концентрических круговых дорожек, как показано на рис. 13.2 (четыре точки указывают четыре считывающие щетки); в этом случае будет получен датчик углового положения. В данном случае, очевидно, требуется применение циклического кода Грея.

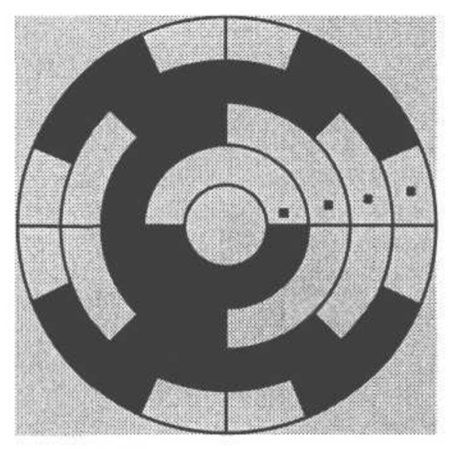


Рис. 13.2. Датчик углового положения

ГЛАВА 14

КРИВАЯ ГИЛЬБЕРТА

В 1890 году **Джузеппе Пеано** (Giuseppe Peano) открыл плоскую **кривую**[1] с удивительным свойством "заполнения пространства". Такая кривая заполняла единичный квадрат и проходила через каждую его точку (x, y) по меньшей мере один раз.

Кривая Пеано основана на разделении каждой стороны единичного квадрата на три равные части, которые делят его на девять меньших квадратов. Кривая проходит эти девять квадратов в определенном порядке. Затем каждый из девяти малых квадратов аналогично делится на девять частей, и кривая модифицируется таким образом, чтобы обойти все части в определенном порядке. Эта кривая может быть описана с использованием дробных чисел, записанных в системе счисления по основанию 3; Пеано описал ее впервые именно так.

В 1891 году Давид Гильберт (David Hilbert) [28] открыл вариант кривой Пеано, основанной на делении каждой стороны единичного квадрата на две равные части, что делит квадрат на четыре меньшие части. Затем каждый из четырех получившихся квадратов, в свою очередь, аналогично делится на четыре меньших квадрата и т.д. На каждой стадии такого деления Гильберт строил кривую, которая обходила все имеющиеся квадраты. Кривая Гильберта (которую иногда называют кривой Пеано-Гильберта) представляет собой предельную кривую, полученную в результате такого построения. Ее можно описать с помощью дробей, записанных в системе счисления по основанию 2.

На рис. 14.1 показаны первые три шага последовательности, приводящей к получению заполняющей пространство кривой Гильберта, в том виде, в котором они были показаны в его статье в 1891 году.

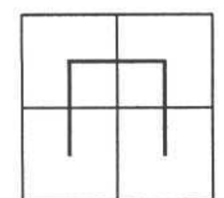
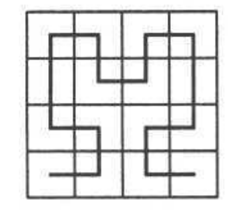
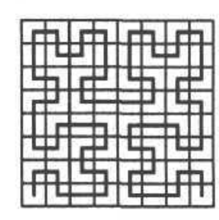
  

Рис. 14.1. Первые три кривые в последовательности, определяющей кривую Гильберта

Здесь мы поступим немного иначе. Используем термин "кривая Гильберта" для любой кривой из последовательности построения заполняющей пространство кривой Гильберта, и под "кривой Гильберта *n*-го порядка" будет подразумеваться *n*-я кривая последовательности (на рис. 14.1 показаны кривые первого, второго и третьего порядков). Каждая кривая порядка *n* масштабируется множителем 2" с тем, чтобы координаты углов кривой представляли собой целые числа. Таким образом, наша кривая Гильберта порядка *n* имеет углы с координатами от 0 до 2" -1 по осям *x* и *y*. Примем положительным направление вдоль кривой от точки $(x, y) = (0,0)$ к $(2'' - 1,0)$. На рис. 14.2 показаны рассматриваемые нами масштабированные кривые Гильберта от первого до шестого порядка.

---

[1] Напомним — *кривая* представляет собой непрерывное отображение одномерного пространства на л-мерное.

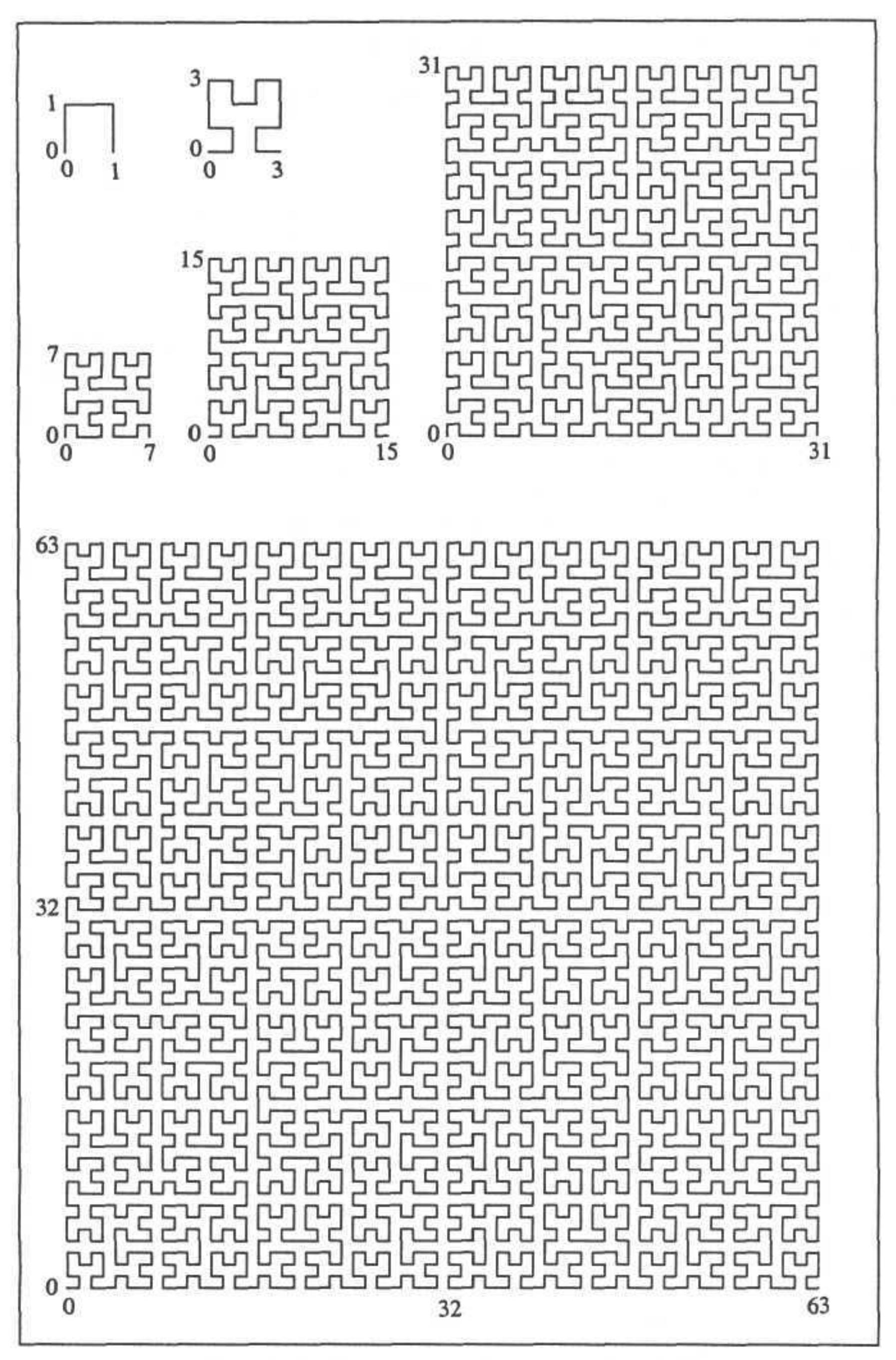


Рис. 14.2. Кривые Гильберта порядка 1–6

14.1. Рекурсивный алгоритм построения кривой Гильберта

Для того чтобы понять, каким образом строится кривая Гильберта, рассмотрим внимательно кривые на рис. 14.2. Кривая порядка 1 состоит из отрезков, направленных вверх, вправо и вниз. Кривая порядка 2 следует тому же общему шаблону. Сначала изображается U-образная кривая, идущая вверх, затем от нее вверх идет отрезок, который соединяется с очередной U-образной кривой, направленной вправо. От нее идет отрезок вправо, к такой же U-образной кривой, от которой отрезок вниз ведет нас к последней U-образной кривой, ведущей вниз. В результате из перевернутой U-образной кривой первого порядка будет получена Y-образная кривая второго порядка.

Кривую Гильберта любого порядка можно рассматривать как серию U-образных кривых разной ориентации, за каждой из которых, за исключением последней, следует отрезок в определенном направлении. При преобразовании кривой Гильберта некоторого порядка в кривую следующего порядка каждая U-образная кривая преобразуется в Y-образную кривую с той же общей ориентацией, а каждый соединяющий отрезок — в отрезок в том же направлении.

Преобразование кривой Гильберта первого порядка (кривая U с общим направлением вправо и вращательной ориентацией по часовой стрелке) в кривую второго порядка происходит следующим образом.

1. Рисуем U по направлению вверх против часовой стрелки.
2. Рисуем отрезок, направленный вверх.
3. Рисуем U по направлению вправо по часовой стрелке.
4. Рисуем отрезок, направленный вправо.
5. Рисуем U по направлению вправо по часовой стрелке.
6. Рисуем отрезок, направленный вниз.
7. Рисуем U по направлению вниз против часовой стрелки.

Рассматривая кривые разных порядков, можно видеть, что все U, ориентированные так же, как и в кривой Гильберта первого порядка, трансформируются таким же образом. Сходный набор правил трансформации может быть создан для каждой U-образной кривой с другой ориентацией. Все эти правила легко укладываются в рекурсивную программу построения кривой Гильберта, показанную в листинге 14.1 [58]. В приведенной программе ориентация U характеризуется двумя целыми числами, которые определяют направление и вращательное направления следующим образом:

dir = 0 : вправо	rot = +1 : по часовой стрелке
dir = 1 : вверх	rot = -1 : против часовой стрелки
dir = 2 : влево	
dir = 3 : вниз	

Листинг 14.1. Генерация кривой Гильберта

```
void step(int);

void hilbert(int dir, int rot, int order)
{

    if (order == 0) return;

    dir = dir + rot;
```

```
    hilbert(dir, -rot, order - 1);
    step(dir);
    dir = dir - rot;
    hilbert(dir, rot, order - 1);
    step(dir);
    hilbert(dir, rot, order - 1);
    dir = dir - rot;
    step(dir);
    hilbert(dir, -rot, order - 1);
}
```

В действительности переменная dir может принимать и другие значения, вне диапазона 0–3, но в этом случае необходимо брать значение dir по модулю 4.

В листинге 14.2 приведена программа-оболочка и функция step, которая используется в функции hilbert. Эта программа получает в качестве аргумента порядок генерируемой кривой Гильберта и выводит список отрезков, указывая для каждого направление перемещения и длину кривой до конца отрезка, а также координаты конца текущего отрезка. Например, для кривой Гильберта второго порядка вывод программы оказывается следующим:

```
 O 0000 00 00
 0 0001 01 00
 1 0010 01 01
 2 OOH  00 01
 1 0100 00 10
 1 0101 00 11
 O OHO  01 11
-1 0111 01 10
 0 1000 10 10
 1 1001 10 11
 O 1010 11 11
-1 1011 11 10
-1 1100 11 01
-2 1101 10 01
-1 1110 10 00
 O 1111 11 00
```

Листинг 14.2. Программа-оболочка для генератора кривой Гильберта

```
#include <stdio.h>
#include <stdlib.h>

int x = -1, y = 0; // Глобальные переменные
int i = 0;          // Расстояние вдоль кривой
int blen;           // Длина вывода

void hilbert(int dir, int rot, int order);

void binary(unsigned k, int len, char *s)
{
    /* Преобразование беззнакового целого k в строку
       символов. Результат хранится в строке s длины len.
     */
    int i;
    s[len] = 0;
```

```
    for (i = len - 1; i >= 0; 1") {
        if (k & 1} s[i] = •1';
        else s[i] = ' 0';
            k = k >> 1;
    }
}

void step(int dir)
{
    char ii[33], xx[17], yy[17];
    switch(dir & 3) {
        case 0: x = x + 1; break;
        case 1: y = y + 1; break;
        case 2: x = x - 1; break;
        case 3: y = y - 1; break;
    }
    binary(i, 2*blen, ii) ,-
    binary(x, blen, xx) ;
    binary(y, blen, yy) ;
    printf("%5d %s %s %s\n", dir, ii, xx, yy);
    i = i + 1; // Увеличение расстояния
}

int main(int argc, char *argv[]) {
    int order;
    order = atoi (argv[1]),•
    blen = order;
    step(0); // Вывод начальной точки
    hilbert(0, 1, order);
    return 0;
}
```

14.2. Преобразование расстояния вдоль кривой Гильберта в координаты

Для того чтобы найти координаты (x, y) точки, определяемой расстоянием s вдоль кривой Гильберта порядка и, заметим, что старшие два бита $2n$-битового числа s определяют, в каком квадранте находится точка. Это происходит потому, что кривая Гильберта любого порядка следует общему шаблону кривой первого порядка. Если два старших бита s равны 00, точка находится где-то в нижнем левом квадранте; 01 указывает на верхний левый квадрант, 10 — правый верхний квадрант, и 11 — нижний правый квадрант. Таким образом, два старших бита s определяют старшие биты n-битовых чисел x и у, как показано ниже.

Два старших бита s	Старшие биты (x, y)
00	(0, 0)
01	(0, 1)
10	(1, 1)
11	(1,0)

В любой кривой Гильберта встречаются только четыре из восьми возможных U-образных кривых. Они графически показаны в табл. 14.1, где приведено также отображение двух битов s на единичные биты x и y.

Таблица 14.1. Четыре возможных отображения

A	B	C	D
П	⊐	⊔	С
00 → (0, 0) 01 → (0, 1) 10 → (1, 1) 11 → (1,0)	00 → (0, 0) 01 → (1,0) 10 → (1, 1) 11 → (0, 1)	00 → (1, 1) 01 → (1, 0) 10 → (0, 0) 11 → (0, 1)	00 → (1, 1) 01 → (0, 1) 10 → (0, 0) 11 → (1 0)

Заметим, что на рис. 14.2 во всех случаях U-образная кривая, представленная отображением A (П) становится на следующем уровне детализации U-образной кривой, представленной отображениями B, A, A или D (в зависимости от расстояния от начала исходного отображения A — O, 1, 2 или 3). Аналогично, U-образная кривая, представленная отображением B (⊐), на следующем уровне детализации становится представленной отображениями A, B, B или C (в зависимости от расстояния вдоль исходной кривой — 0, 1, 2 или 3). Результатом таких наблюдений является табл. 14.2, в которой представлены переходы между состояниями, соответствующими отображениям, показанным в табл. 14.1.

Таблица 14.2. Таблица переходов состояний для вычисления (x, y) из s

Еслитекущее состояние	и следующие (справа) биты s	то добавляем к (x,y)	и переходим ксостоянию
A	00	(0, 0)	B
A	01	(0, 1)	A
A	10	(1, 1)	A
A	11	(1,0)	D
B	00	(0, 0)	A
B	01	(1,0)	B
B	10	(1, 1)	B
B	11	(0, 1)	C
C	00	(1, 1)	D
C	01	(1,0)	C
C	10	(0, 0)	C
C	11	(0, 1)	B
D	00	(1, 1)	C
D	01	(0, 1)	D
D	10	(0, 0)	D
D	11	(1,0)	A

Для использования этой таблицы начнем с состояния A. Целое s следует дополнить необходимым количеством нулевых битов слева с тем, чтобы его длина стала равной $2n$ битов, где я — порядок кривой Гильберта. После этого сканируем попарно биты s слева направо. Первая строка табл. 14.2 означает, что если текущим состоянием является A и сканированная пара битов s равна 00, то мы получаем на выходе (0,0) и переходим к со-

стоянию B, после чего считываем очередные два бита слова *s*. Аналогично, вторая строка табл. 14.2 означает, что если текущим состоянием является A, а сканированная пара битов *s* равна 01, то на выходе мы получаем $(0,1)$ и остаемся в состоянии A.

Получаемые на выходе биты накапливаются в порядке слева направо. Когда *s* оказывается сканированным до конца, мы получаем на выходе *n*-битовые величины *x* и *y*. Предположим в качестве примера, что $n=3$ и $s=110100$. Поскольку процесс начинается в состоянии A и первые сканированные биты равны 11, мы получаем на выходе $(1,0)$ и переходим в состояние D (четвертая строка табл. 14.2). Затем, находясь в состоянии D и получая очередную пару битов *s*, равную 01, в соответствии с 14-й строкой табл. 14.2 мы получаем на выходе $(0,1)$ и остаемся в состоянии D. И наконец, последняя пара битов 00 дает нам на выходе $(1,1)$ и переводит нас в состояние C (хотя, поскольку строка *s* сканирована до конца, это состояние не играет никакой роли).

Итак, в результате сканирования и переходов получена пара $(101,011)$, т.е. $x=5$ и $y=3$.

Реализующая описанный алгоритм программа на языке программирования C приведена в листинге 14.3. В этой программе текущее состояние представлено целым числом из диапазона 0–3, соответствующего состояниям A-D. В присвоении переменной row текущее состояние объединяется с очередными двумя битами s, давая целое число от О до 15, которое представляет собой номер строки в табл. 14.2. Переменная row используется при обращении к целым числам, которые представляют собой два правых столбца табл. 14.2, т.е., по сути, осуществляется табличный поиск в регистре. В используемых шестнадцатеричных значениях биты слева направо соответствуют значениям табл. 14.2, просматриваемым снизу вверх.

Листинг 14.3. Программа для вычисления *x* и *y* по значению *s*

```
void hil_xy_from_s(unsigned s,   int n,
                   unsigned *xp, unsigned *yp)
{
    int i;
    unsigned state, x, y, row;
    state = 0;                          // Инициализация
    x = y = 0;
    for (i = 2*n - 2; i >= 0; i -= 2)  // Выполнить п раз
    {
        row = 4*state | (s >> i) & 3;  // Строка в таблице
        x = (x << 1) | (0x936C >> row) & 1;
        y = (y << 1) | (0x39C6 >> row) & 1;
        state =
            (Ox3E6B94C1 >> 2*row) & 3; // Новое состояние
    }
    *xp = x;   // Возврат
    *yp = У;   // результатов
}
```

В [44] приводится несколько иной алгоритм. В отличие от алгоритма из листинга 14.3, он сканирует биты *s* справа налево. Алгоритм базируется на том, что можно отобразить младшие значащие биты *s* на (x,y), основываясь на кривой Гильберта первого порядка, после чего переходить к тестированию следующей пары битов *s*. Если они равны 00, то только что вычисленные значения (x,y) должны поменяться местами (что соответствует отражению относительно прямой $x=y$ (см. рис. 14.1, кривые первого и второго порядка). Если эти биты равны 01 или 10, то значения *x* и *y* остаются неизменными.

И наконец, если значение пары битов равно 11, то значения x и y обмениваются и к ним применяется операция дополнения. Те же правила применяются и при дальнейшем сканировании битовой строки s. Этот алгоритм схематично представлен в табл. 14.3, а реализующий его код — в листинге 14.4. Интересно, что сначала можно добавлять биты к x к y, а уже затем выполнять операции обмена и дополнения над полученными значениями, включающими только что добавленные биты; результат окажется тем же, что и при добавлении битов после обмена и дополнения.

Таблица 14.3. Метод вычисления *(x,y)* по значению *s* [44]

Если следующие два бита s слева равны	то	и добавляем спереди к (x,y)
00	Обмениваем x и y	(0, 0)
01	Оставляем x и y без изменений	(0, 1)
10	Оставляем x и y без изменений	(1, 1)
11	Обмениваем x и y и выполняем операцию дополнения	(1, 0)

Листинг 14.4. Метод вычисления *(x,y)* по значению *s* [44]

```
void hil_xy_from_s(unsigned s, int n,
                   unsigned *xp, unsigned *yp)
{
    int i, sa, sb;
    unsigned x, y, temp;
    for (i = 0; i < 2*n; i += 2) {
        sa = (s >> (i+1)) & 1;     // Бит i+1 строки s
        sb = (s >> i) & 1;         // Бит i   строки s
        if ((sa ^sb) == 0)
        {                          // Если sa,sb = 00 или 11,
            temp = x;              // обмениваем x и y
            x = y^(-sa);           // и, если sa = 1,
            y = temp^(-sa);        // дополняем их
        }
        x = (x>>1) |
            (sa << 31);            // Добавляем sa к x и
        y = (y >> 1) |
            ((sa ^ sb) << 31);     // (sa^sb) к y (спереди)
    }
    *xp = x >> (32 - n);           // Выравниваем x и y вправо
    *yp = y >> (32 - n);           // и возвращаем их
}
```

Переменные x и y в листинге 14.4 не инициализированы, что в некоторых компиляторах может привести к сообщению об ошибке. Однако приведенный код корректно функционирует независимо от того, какие значения имели переменные x и y изначально.

Условного перехода в цикле в листинге 14.4 можно избежать, если воспользоваться рассматривавшимся в разделе 2.19 приемом "*трех исключающих или*". Блок if при этом можно заменить следующим кодом (здесь swap и cmpl — беззнаковые целые переменные):

```
swap =  (sa ^sb)  - 1;
    // Если требуется обмен,  -1;  в противном случае 0

cmpl  =  -(sa & sb);
    // Если требуется дополнение,  -1;  в противном случае 0

X = X ^y;
y = y    (x & swap)  ^ cmpl;
x = x ^y;
```

Однако при этом требуется выполнение девяти команд, в то время как при использовании условного перехода достаточно только двух или шести, в зависимости от выполнения условия, так что, возможно, отказ от условного перехода в данном случае — не лучший выбор.

Идея "обмена и дополнения" из [44] подсказывает логическую схему для построения кривой Гильберта. Идея, лежащая в основе описываемой далее схемы, состоит в том, чтобы, проходя вдоль кривой n-го порядка, вы отображали пары битов s в координаты (x, y) в соответствии с отображением А из табл. 14.1. При прохождении различных областей координаты, полученные в результате отображения, могут меняться местами, дополняться либо над ними могут выполняться обе эти операции. Схема, показанная на рис. 14.3, отслеживает необходимость выполнения данных операций на каждом шаге, использует соответствующее отображение двух битов s на (x_i, y_i) и генерирует сигналы обмена и дополнения для следующего шага.

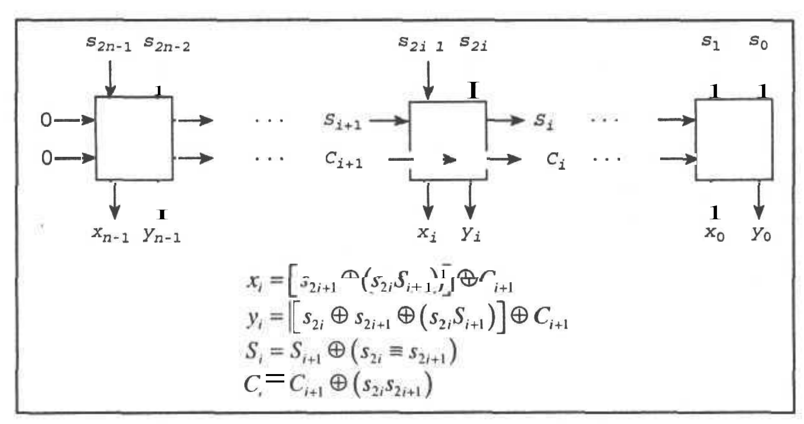


Рис. 14.3. Логическая схема для вычисления (x, y) по значению s

Предположим, что имеется регистр, содержащий длину пути s вдоль кривой Гильберта и схемы для ее увеличения. Тогда для поиска следующей точки кривой Гильберта следует сначала увеличить значение s, а затем преобразовать его так, как показано в табл. 14.4. Преобразования s в таблице выполняются слева направо, что создает определенные трудности, так как увеличение s — процесс, затрагивающий биты в порядке справа налево. Таким образом, время, необходимое для генерации очередной точки кривой Гильберта n-го порядка, пропорционально 2л (для увеличения л) плюс n (для преобразования 5 в пару координат (x, y)).

Таблица 14.4. Логика вычислений (x,y) по значению s

Если очередная (справа) пара битов s равна	то добавляем к (x,y)	и устанавливаем
00	(0, 0)*	$swap = \overline{swap}$
01	(0, 1)*	Без изменений
10	(1, 1)*	Без изменений
11	(1, 0)*	$swap = \overline{swap}$, $cmpl = \overline{cmpl}$
* Возможно, обмененные или дополненные		

На рис. 14.3 эти вычисления показаны в виде логической схемы (S обозначает сигнал обмена, а C — сигнал дополнения).

В схеме на рис. 14.3 предлагается другой путь вычисления (x,y) по значению 5. Обратите внимание на то, как сигналы обмена и дополнения распространяются слева направо через п стадий. Таким образом, здесь можно применить метод параллельного префикса для быстрого (не за п -1, а за $\log_2 n$ шагов) распространения информации об обмене и дополнении по всем стадиям, а затем использовать некоторые побитовые операции для вычисления x и y с использованием приведенных на рис. 14.3 уравнений. Значения x и y перемешаны, и их биты находятся в четных и нечетных позициях слова, так что их следует извлечь оттуда с использованием информации из раздела 7.2. Это может показаться несколько сложным и окупающимся только для больших значений n, однако давайте посмотрим, как все происходит на самом деле.

Процедура, реализующая описанную операцию, приведена в листинге 14.5 [19]. Эта процедура оперирует с величинами, представляющими собой полные слова, так что сначала входное значение s дополняется слева битами '0Г (эта битовая комбинация не влияет на количество обменов и дополнений). Затем вычисляется величина cs (complement-swap, дополнение-обмен). Это слово имеет вид `cscs...cs`, где каждый отдельный бит с, будучи 1, означает, что соответствующая пара битов должна быть дополнена, а s — что соответствующая пара битов должна быть обменена в соответствии с табл. 14.3. Другими словами, каждая пара битов s отображается в пару cs следующим образом:

s_{2i+1}	s_{2i}	cs
0	0	01
0	1	00
1	0	00
1	1	11

Это и есть та величина, к которой мы хотим применить операцию параллельного префикса, а именно PP-XOR. Операция применяется слева направо, поскольку последовательные единичные биты, означающие дополнение или обмен, имеют те же логические свойства, что и операция *исключающего или:* два последовательных единичных бита взаимно исключают друг друга.

Листинг 14.5. Метод параллельного префикса для вычисления (x,y) по значению s

```
void hil_xy_from_s(unsigned s, int n,
                   unsigned *xp, unsigned *yp)
{
```

```
    unsigned comp, swap, cs, t, sr;
    s = s | (0x55555555 << 2*n); // Заполняем s слева парами
    sr = (s >> 1) & 0x55555555;  // 01 (без изменений)
    cs = ((s & 0x55555555) + sr) // Вычисляем пары дополнений
        ^ 0x55555555;            // и обменов
    // Операция PP-XOR распространяет информацию о дополнениях
    // и обменах слева направо, парами (шаг cs ^= cs >> 1
    // отсутствует), что дает в результате две операции
    // параллельного префикса над двумя чередующимися
    // множествами из 16 битов каждое
    cs = cs ^ (cs >> 2) ;
    cs = cs ^(cs >> 4);
    cs = cs ^(cs >> 8);
    cs = cs ^(cs >>16) ;
    swap = cs & 0x55555555;        // Разделяем биты обмена
    comp = (cs >> 1) & 0x55555555; // и дополнения
    t = (s & swap) ^ comp;         // Вычисляем x и y в
    s = s ^sr ^t ^(t <<1) ;        // нечетных и четных
                                   // позициях
    s = s & ((1 << 2*n) - 1);      // Убираем лишние биты слева
    // Разделяем значения x и y
    t = (s ^ (s >> 1)) & 0x22222222; s = s * t ^(t << 1);
    t = (s ^ (s >> 2)) & 0x0C0C0C0C; s = s * t ^ (t <<2) ;
    t = (s ^ (s >> 4)) & 0x00F000F0; s = s ^ t ^ (t <<4);
    t = (s ^ (s >> 8)) & 0x0000FF00; s = S ^t * (t <<8);
    *xp = s >> 16;    // Присваиваем переменным значения
    *yp = s & 0xFFFF; // x и y
}
```

Оба сигнала (дополнения и обмена) распространяются одной и той же операцией PP-XOR.

Следующие четыре присвоения транслируют каждую пару битов s в значения (x,y), где x состоит из битов, находящихся в нечетных (левых) позициях, а y — из битов в четных позициях. Хотя сама логика может показаться и непонятной, не так уж сложно убедиться в том, что каждая пара битов s преобразуется в соответствии с первыми двумя уравнениями на рис. 14.3 (указание: рассмотрите отдельно преобразования битов в четных и нечетных позициях, не забывая о том, что в нечетных позициях биты t и sr равны 0).

Оставшаяся часть процедуры очевидна. Всего она требует выполнения 66 базовых RISC-команд (без ветвлений), в то время как код из листинга 14.2 требует в среднем выполнения $19n+10$ команд (вычислено на основе скомпилированного кода). Таким образом, уже при $n>3$ метод параллельного префикса оказывается более быстрым.

14.3. Преобразование координат в расстояние вдоль кривой Гильберта

Если заданы координаты точки на кривой Гильберта, то расстояние до данной точки вдоль кривой от ее начала можно вычислить при помощи таблицы переходов состояний, подобной табл. 14.2. Для нашей задачи такой таблицей переходов является табл. 14.5.

Таблица 14.5. Таблица переходов состояний для вычисления *s* по значениям *(x,y)*

Если текущее состояние	и следующие справа два бита *(x,y)* равны	то добавляем к *s*	и переходим в состояние
A	(0, 0)	00	B
A	(0, 1)	01	A
A	(1,0)	11	D
A	(1, 1)	10	A
B	(0, 0)	00	A
B	(0, 1)	11	C
B	(1,0)	01	B
B	(1,1)	10	B
C	(0, 0)	10	C
C	(0, 1)	11	B
C	(1, 0)	01	C
C	(1, 1)	00	D
D	(0, 0)	10	D
D	(0, 1)	01	D
D	(1,0)	11	A
D	(1, 1)	00	C

Работа с этой таблицей аналогична работе с таблицей переходов состояний в предыдущем разделе. Сначала значения x и y должны быть дополнены слева ведущими нулевыми битами с тем, чтобы длина битовых строк, представляющих эти величины, стала равной n, где n — порядок рассматриваемой кривой Гильберта. Затем биты x и y сканируются слева направо, и значение s строится в том же направлении.

Программа на языке программирования С, реализующая эти действия, представлена в листинге 14.6.

Листинг 14.6. Программа для вычисления *s* по значениям *(x,y)*

```
unsigned hil_s_from_xy(unsigned x,
                       unsigned y, int n)
{
    int i;
    unsigned state, s, row;
    state = 0;                  // Инициализация
    s = 0;
    for (i = n - 1; i >= 0; i--) {
        row = 4*state | 2*((x >> i) & 1) | (y>> i) & 1;
        s = (s << 2) | (0x361E9CB4 >> 2*row) & 3;
        state = (0x8FE65831 >> 2*row) & 3;
    }
    return s;
}
```

В [44] приведен алгоритм для вычисления s по значениям (x,y), похожий на соответствующий алгоритм для вычислений в обратном направлении — (x,y) по значению s. Этот алгоритм основан на сканировании слева направо и показан в табл. 14.6 и листинге 14.7.

Таблица 14.6. Метод вычисления s по значениям (x,y) [44]

Если следующие два бита (x,y) справа равны	то	и добавляем к s
(0, 0)	Обмениваем x и y	00
(0, 1)	Оставляем x и y без изменений	01
(1,0)	Обмениваем x и y и выполняем операцию дополнения	11
(1, 1)	Оставляем x и y без изменений	10

Листинг 14.7. Метод вычисления s по значениям (x,y) [44]

```
unsigned hil_s_from_xy(unsigned x,
                       unsigned y, int n)
{
    int i, xi, yi;
    unsigned s, temp;
    s = 0;                        // Инициализация
    for (i = n - 1; i >= 0; i--) {
        xi = (x >> i) & 1;        // Получаем i-й бит x
        yi = (y >> i) & 1;        // Получаем i-й бит y
        if (yi == 0) {
            temp = x;             // Обмениваем x и y и,
            x = y^(-xi);          // если xi = 1,
            y = temp^(-xi);       // дополняем их
        }
        s = 4*s + 2*xi + (xi^yi); // Добавляем два бита к s
    }
    return s;
}
```

14.4. Увеличение координат кривой Гильберта

Пусть заданы координаты (x,y) точки на кривой Гильберта n-го порядка. Каким образом можно найти координаты следующей точки? Простейший путь, конечно, состоит в преобразовании координат в расстояние вдоль кривой s, увеличении s на 1 и обратном преобразовании в координаты (x,y) с использованием описанных в предыдущих разделах алгоритмов.

Более удачный (но не слишком) путь основывается на том факте, что при перемещении вдоль кривой Гильберта на каждом шаге или x, или y, но не обе величины одновременно либо увеличиваются, либо уменьшаются на 1. Поэтому можно разработать алгоритм, который сканирует координаты слева направо для определения того, какой тип U-образной кривой представляют собой младшие два бита. Затем на основании этой информации и значения двух младших битов осуществляется увеличение или уменьшение x или y.

В данном алгоритме есть сложность, возникающая на каждом четвертом шаге, когда рассматриваемая точка оказывается в конце U-образной кривой. В этой точке

направление определяется *предыдущими* битами x и y и U-образной кривой более высокого порядка, с которой эти биты связаны. Если же точка на U-образной кривой более высокого порядка также оказывается в конце, то необходимо в очередной раз рассмотреть предыдущие биты x и y и U-образную кривую еще более высокого порядка, и т.д.

Этот алгоритм описывается табл. 14.7, где A, B, C и D обозначают U-образные кривые, показанные в табл. 14.1. Для использования данной таблицы сначала надо добавить к x и y ведущие нулевые биты с тем, чтобы получились и-битовые значения (где n — порядок рассматриваемой кривой Гильберта). Начиная с состояния A, сканируем биты x и y слева направо. Первая строка табл. 14.7 означает, что если текущим состоянием является A, а сканированы биты (0,0), то надо установить переменную-флаг, указывающую необходимость увеличения y, и перейти в состояние B. Прочие строки таблицы интерпретируются аналогично; суффикс "минус" означает, что соответствующая координата должна быть уменьшена. Прочерк в третьем столбце таблицы говорит о том, что переменная-флаг, отслеживающая изменение координат, остается на данном шаге неизменной.

По окончании сканирования битов x и y увеличиваем (или уменьшаем) значение одной из координат в соответствии со значением переменной-флага.

Таблица 14.7. Добавление одного звена кривой Гильберта

Если текущее состояние	**а очередные два бита *(x,y)* справа**	**то нужно изменить координату**	**и перейти в состояние**
A	(0, 0)	*y*+	**B**
A	(0, 1)	*x*+	**A**
A	**(1,0)**	—	**D**
A	(1, 1)	*y*–	**A**
B	(0, 0)	*x*+	**A**
B	(0, 1)	—	**C**
B	**(1,0)**	*y*+	**B**
B	(1, 1)	*x*-	**B**
C	(0, 0)	*y*+	**C**
C	(0, 1)	—	**B**
C	**(1,0)**	*x*-	**C**
C	(1, 1)	*y*–	**D**
D	(0, 0)	*x*+	**D**
D	(0, 1)	*y*–	**D**
D	**(1,0)**	—	**A**
D	(1, 1)	*x*-	**C**

Программа на языке C, реализующая рассмотренный алгоритм, представлена в листинге 14.8. Переменная `dx` инициализируется таким образом, чтобы при многократных вызовах данной функции в цикле генерировалась одна кривая Гильберта.

Листинг 14.8. Программа для добавления одного звена кривой Гильберта

```
void hil_inc_xy(unsigned *xp, unsigned *yp, int n)
{
    int i;
    unsigned x, y, state, dx, dy, row, dochange;
    x = *xp;
    y = *yp;
    state = 0;            // Инициализация
    dx = -((1 << n) - 1); // -(2**n - 1)
    dy = 0;
    for (i = n-1; i >= 0; i--) {   // Выполняем n раз
        row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
        dochange = (0xBDDB >> row) & 1;
        if (dochange) {
            dx = ((0x16451659 >> 2*row) & 3) - 1;
            dy = ((0x51166516 >> 2*row) & 3) - 1;
        }
        state = (0X8FE65831 >> 2*row) & 3;
    }
    *xp = *xp + dx;
    *yp = *yp + dy;
}
```

Табл. 14.7 легко реализуется представленной на рис. 14.4 логикой. На этом рисунке использованы приведенные ниже обозначения.

- x_i i-й бит входного значения x
- y_i i-й бит входного значения y
- X, Y Обмененные и дополненные в соответствии со значениями S_{i+1} и C_{i+1} значения x_i и y_i
- I Флаг: если равен 1 — увеличение на 1, если 0 — уменьшение на 1
- W Флаг: если равен 1, увеличивается или уменьшается x, если 0 — увеличивается или уменьшается y
- S Если значение S равно 1, выполняется обмен x_i и y_i
- C Если значение C равно 1, выполняется дополнение x_i и y_i

Пары значений S и C определяют состояние в табл. 14.7: пары (S,C), равные (0,0), $(0,1)$, (1,0) и $(1,1)$, обозначают соответственно состояния A, B, C и D. Выходные сигналы I_0 и Wo указывают, должно ли выполняться уменьшение или увеличение координаты и какой именно. (В дополнение к указанной на рисунке логике требуется схема увеличения/уменьшения с мультиплексорами, указывающими, какое именно значение — x или y — должно быть увеличено или уменьшено, а также схема, которая поместит вычисленное значение назад в регистр, где хранится переменная x или y. В качестве альтернативы можно обойтись двумя схемами увеличения/уменьшения.)

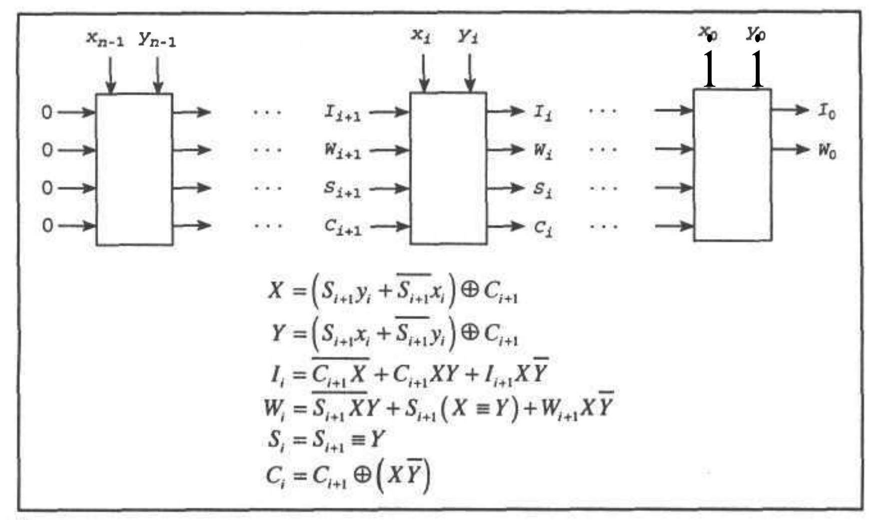


Рис. 14.4. Логическая схема для добавления одного звена кривой Гильберта

14.5. Нерекурсивный алгоритм генерации кривой Гильберта

Алгоритмы, приведенные в табл. 14.2 и 14.7, предоставляют два нерекурсивных алгоритма построения кривой Гильберта любого порядка. Каждый из них без особых сложностей может быть реализован аппаратно. Аппаратное обеспечение для реализации алгоритма на базе табл. 14.2 включает регистр для хранения величины s, увеличивающейся на каждом шаге построения кривой и преобразуемой в координаты (x, y). Аппаратное обеспечение, реализующее алгоритм на основе табл. 14.7, не требует регистра для хранения s, но используемый при этом алгоритм более сложен.

14.6. Другие кривые, заполняющие пространство

Как упоминалось, Пеано был первым, открывшим в 1890 году кривую, заполняющую пространство. С тех пор открыто множество других кривых, которые часто носят общее название "кривые Пеано". В 1900 году Муром (Eliakim Hastings Moore) была открыта интересная вариация кривой Гильберта. Она "циклична" в том смысле, что ее конечная точка отстоит на один шаг от начальной. На рис. 14.5 показаны кривая Пеано третьего порядка и кривая Мура четвертого порядка. Кривая Мура иррегулярна в том, что кривая первого порядка представляет собой кривую "вверх-вправо-вниз" (⊓↓), но данная U-образность в кривых более высокого порядка отсутствует. За этим малым исключением алгоритмы для работы с кривой Мура очень схожи с соответствующими алгоритмами для кривой Гильберта.

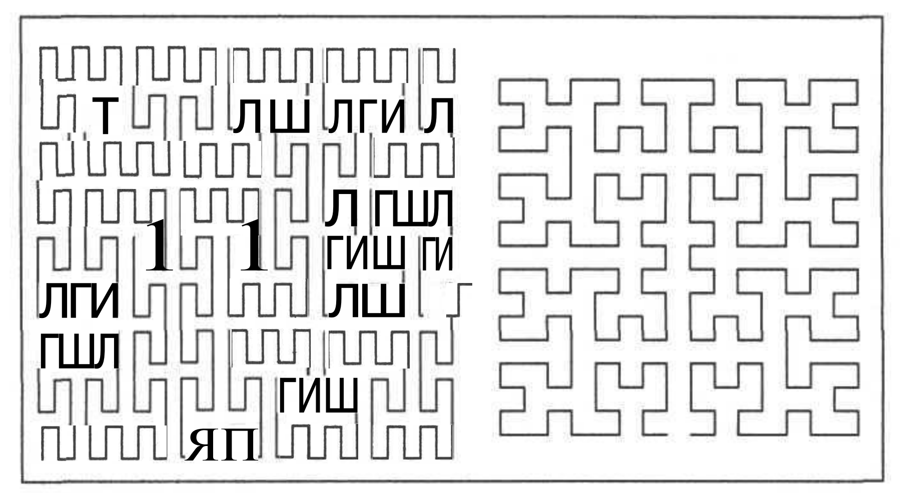
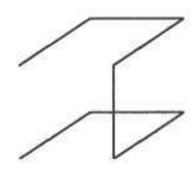


Рис. 14.5. Кривые Пеано (слева) и Мура (справа)

Кривая Гильберта обобщена для произвольных прямоугольников на три и большее число размерностей. Базовый строительный блок трехмерной кривой Гильберта показан ниже. Он проходит по всем точкам куба 2x2x2. Эти и множество других кривых, заполняющих пространство, рассматриваются в работе [55].



14.7. Применение

Кривые, заполняющие пространство, находят применение при обработке изображений — сжатии, формировании полутонового изображения и текстурном анализе [44]. Еще одно приложение состоит в повышении производительности трассировки лучей и графической визуализации. Условно сцена сканируется лучом, который перемещается в соответствии с обычным растром — слева направо через весь экран с перемещением сверху вниз. При этом, когда луч попадает на объект в базе данных моделируемой сцены, определяются его цвет и прочие свойства и изменяется вид (цвет и яркость) соответствующего пикселя на экране. (Конечно, это сверхупрощенное описание, но для наших целей его вполне достаточно.) Одна из проблем состоит в том, что зачастую такая база данных достаточно велика и данные по каждому объекту должны загружаться в память, когда сканирующий луч попадает на него. При построчном сканировании обычна ситуация, когда при проходе по строке приходится загружать в память те же объекты, которые загружались при предыдущем сканировании. Количество загрузок в память можно резко снизить, если сканирование будет обладать свойством локализации, например сканирование по квадрантам (когда переход к сканированию очередного квадранта происходит только по завершению сканирования предыдущего) зачастую позволяет существенно снизить количество загрузок в память информации об одном объекте.

Кривая Гильберта, пожалуй, обладает искомым свойством локализации. Она рекурсивно сканирует квадрант за квадрантом и не делает длинных переходов из одного квадранта в другой.

Дуглас Вурхис (Douglas Voorhies) [58] смоделировал поведение загрузки информации об объектах в память для обычного однонаправленного сканирования, кривой Пеано и кривой Гильберта. Его метод состоял в случайном размещении на экране окружностей заданного размера. Когда путь сканирования попадает в окружность, представляющую новый объект, последний загружается в память. Когда путь сканирования покидает объект, информация о нем не выгружается из памяти до тех пор, пока путь сканирования не отойдет от объекта на расстояние, равное удвоенному радиусу. Таким образом, если путь сканирования удаляется недалеко от объекта и вновь возвращается к нему, операции выгрузки и загрузки в память не происходит. Вурхис проводил эксперименты для окружностей разного радиуса на экране размером 1024x1024.

Считаем, что каждый раз, когда путь сканирования попадает в объект и покидает его, происходит одна операция загрузки в память. Тогда очевидно, что обычное построчное сканирование одной окружности диаметра D приводит к D операциям загрузки в память. Интересным результатом моделирования Вурхиса оказалось то, что для кривой Пеано количество операций загрузки в память в среднем составляет 2.7 и не зависит от диаметра окружности. Для кривой Гильберта среднее количество операций загрузки в память также не зависит от диаметра окружности, но составляет около 1.4. Таким образом, экспериментально доказано, что в смысле локализации при сканировании кривая Гильберта превосходит кривую Пеано и обе они существенно превосходят построчное сканирование.

ГЛАВА 15

ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ

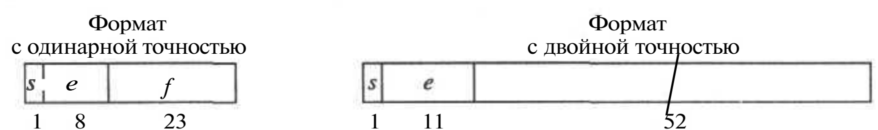
Бог создал целые числа;
все остальные —
дело рук человеческих.
Леопольд Кронекер
(Leopold Kronecker)

Выполнение операций над числами с плавающей точкой с помощью целочисленной арифметики и логических команд — предложение далеко не лучшее. Особенно это относится к правилам и форматам стандарта *IEEE Standard for Binary Floating-Point Arithmetic,* ШЕЕ Std. 754-1985, общеизвестного как "IEEE-арифметика". В этой арифметике имеется число NaN (not a number, не число) и бесконечности, которые представляют собой частные случаи практически для всех операций. В ней есть плюс ноль и минус ноль, которые должны быть эквивалентны при сравнении, кстати имеющем в IEEE-арифметике четвертый возможный результат — "неупорядочено". Старший значащий бит дроби в "нормальных" числах явно не участвует, но участвует в "денормализованных" и "субнормальных" числах. Дробные части представлены в виде обычных чисел со знаком, в то время как показатели степени — в смещенном виде. На все это, конечно, есть свои причины, но в результате получаются программы, заполненные условными переходами, которые очень сложно эффективно реализовать.

В данной главе предполагается, что читатель немного знаком со стандартом IEEE, так что о нем будет сказано предельно коротко.

15.1. Формат IEEE

Ограничимся рассмотрением чисел одинарной и двойной точности (32 и 64 бита), описанных в IEEE 754. В стандарте представлены также "расширенный одинарный" и "расширенный двойной" форматы, но их описание весьма неточно в связи с тем, что ряд деталей зависит от реализации (например, ширина показателя степени не определена стандартом). Ниже показаны одинарный и двойной форматы чисел с плавающей точкой.



Знаковый бит равен 0 для положительных чисел и 1 для отрицательных. Смещенный показатель степени *e* и дробная часть / представляют собой величины, старший бит которых находится слева. Числа с плавающей точкой кодируются в соответствии со стандартом IEEE так, как показано ниже.

Одинарная точность				Двойная точность		
e	*f*	*Значение*		*e*	*f*	*Значение*
0	0	± 0		0	0	± 0
0	$\neq 0$	$\pm 2^{-126}(0.f)$		0	$\neq 0$	$\pm 2^{-1022}(0.f)$
от 1 до 254	–	$\pm 2^{e-127}(1.f)$		от 1 до 2046	–	$\pm 2^{e-1023}(1.f)$
255	0	$\pm\infty$		2047	0	$\pm\infty$
255	$\neq 0$	NaN		2047	$\neq 0$	NaN

В качестве примера рассмотрим кодирование числа л в формате с одинарной точностью. В соответствии с [38], в двоичном представлении

я» 11.0010 0100 0011 1111 0110 1010 1000 1000 10000101 10100011 0000 10...

Эта величина находится в диапазоне "нормализованных" чисел, представленном третьей строкой приведенной выше таблицы. Старший единичный бит числа я опускается, поскольку ведущий единичный бит не хранится при кодировании нормализованных чисел. Чтобы двоичная точка размещалась в правильном месте, показатель степени $e-127$ должен быть равен 1, так что $e=128$. Таким образом, представление числа я выглядит как

О 10000000 10010010000111111011011

или в шестнадцатеричной записи:

40490FDB,

где дробь округлена до ближайшего представимого числа.

Числа, для которых $1 \le e \le 254$, называются "нормализованными". Старший бит таких чисел не хранится явно. Ненулевые числа с $e=0$ называются "денормализованными", и их старший бит хранится в представлении числа в явном виде. Такая схема иногда называется схемой с "постепенной потерей значимости" (gradual underflow). В табл. 15.1 приведены некоторые крайние значения из различных диапазонов чисел с плавающей точкой. "Максимальное целое" в приведенной таблице означает наибольшее целое число, такое, что все целые числа, не превосходящие его по абсолютному значению, представимы в виде чисел с плавающей точкой точно; следующее за ним число уже должно быть округлено.

Таблица 15.1. Некоторые предельные значения чисел с плавающей точкой

	Шестнадцатеричное представление	Точное значение	Приближенное значение
Одинарная точность			
Наименьшее денормализованное	0000 0001	2^{-149}	1.401×10^{-45}
Наибольшее денормализованное	007F FFFF	$2^{-126}(1-2^{-23})$	1.175×10^{-38}
Наименьшее нормализованное	0080 0000	2-126	1.175×10^{-38}
1.0	3F80 0000	1	1
Максимальное целое	4B80 0000	2^{24}	1.677×10^{7}

Окончание табл. 15.1

	Шестнадцатеричное представление	Точное значение	Приближенное значение
Одинарная точность			
Наибольшее нормализованное	`7F7F FFFF`	$2^{128}\left(1-2^{-24}\right)$	3.403×10^{38}
∞	`7F80 0000`	∞	∞
Двойная точность			
Наименьшее **денормализованное**	`0...0001`	2^{-1074}	4.941×10^{-324}
Наибольшее денормализованное	`000F...F`	$2^{-1023}\left(1-2^{-52}\right)$	2.225×10^{-308}
Наименьшее нормализованное	`0010...0`	2^{-1022}	2.225×10^{-308}
1.0	`3FF0...0`	1	1
Максимальное целое	`4340...0`	2^{53}	9.007×10^{15}
Наибольшее нормализованное	`7EEF...F`	$2^{1024}\left(1-2^{-53}\right)$	1.798×10^{308}
∞	`7FF0...0`	∞	∞

Для нормализованных чисел единица в последней позиции представляет собой относительное значение в диапазоне от $1/2^{24}$ до $1/2^{23}$ (примерно от 5.96×10^{-8} до 1.19×10^{-7}) для одинарной точности и от $1/2^{53}$ до $1/2^{52}$ (примерно от 1.11×10^{-16} до 2.22×10^{-16}) для двойной точности. Максимальная "относительная ошибка" при округлении к ближайшему числу составляет половину приведенной величины.

Диапазон целых чисел, точно представимых в формате с плавающей точкой, составляет от -2^{24} до $+2^{24}$ (т.е. от -16777216 до $+16777216$) для одинарной точности и от -2^{53} до $+2^{53}$ (от -9007199254740992 до $+9007199254740992$) для двойной точности. Конечно, некоторые целые числа вне этого диапазона также могут быть представлены точно, например большие степени 2; однако указанные диапазоны являются максимальными диапазонами, *все* числа которых представимы точно.

Замена деления на константу умножением может быть выполнена с полной (ШЕЕ) точностью только для чисел, для которых соответствующие множители могут быть точно представлены в формате с плавающей точкой. Это степени 2 от -127 до 127 для одинарной точности и от -1023 до 1023 для двойной. Заметим, что числа 2^{-127} и 2^{-1023} являются **денормализованными**, так что их использования лучше избегать на машинах, которые недостаточно эффективно реализуют операции над денормализованными числами.

15.2. Сравнение чисел с плавающей точкой с использованием целых операций

Одним из достоинств IEEE-кодирования чисел с плавающей точкой является то, что все не NaN-значения корректно упорядочены, если рассматривать их как знаковые целые числа.

Для того чтобы запрограммировать сравнение чисел с плавающей точкой с использованием целых операций, необходимо отказаться от результата сравнения "неупорядочено". В IEEE 754 такой результат получается в том случае, если один или оба аргумента оператора сравнения представляют собой значение NaN. Описываемый далее метод трактует значения NaN как числа, превышающие бесконечное значение.

Сравнение становится существенно проще, если считать, что величина -0.0 строго меньше +0.0 (что не согласуется со стандартом ШЕЕ 754). Считая описанные допущения приемлемыми и используя для сравнений с плавающей точкой обозначения $\overset{f}{<}$, $\overset{f}{\le}$ и $\overset{f}{=}$, а символ $\approx$ — как напоминание о том, что приведенные формулы не совсем корректно работают со значениями ± 0.0, можно использовать следующие формулы:

$$a \overset{f}{=} b \approx (a = b),$$

$$a \overset{f}{<} b \approx (a \ge 0 \,\&\, a < b) \,|\, (a < 0 \,\&\, a \overset{u}{>} b),$$

$$a \overset{f}{\le} b \approx (a \ge 0 \,\&\, a \le b) \,|\, (a < 0 \,\&\, a \overset{u}{\ge} b).$$

Если же значение -0.0 в обязательном порядке должно быть равно значению +0.0, то приведенные формулы существенно усложняются (хотя и не настолько, как могло бы показаться необходимым на первый взгляд).

$$\begin{aligned} a \overset{f}{=} b &\equiv (a = b) \,|\, (-a = a \,\&\, -b = b) \\ &\equiv (a = b) \,|\, ((a \,|\, b) = \text{0x80000000}) \\ &\equiv (a = b) \,|\, (((a \,|\, b) \,\&\, \text{0x7FFFFFFF}) = 0) \end{aligned}$$

$$a \overset{f}{<} b \equiv \left((a \ge 0 \,\&\, a < b) \,|\, (a < 0 \,\&\, a \overset{u}{>} b)\right) \,\&\, \left((a \,|\, b) \ne \text{0x80000000}\right)$$

$$a \overset{f}{\le} b \equiv (a > 0 \,\&\, a \le b) \,|\, (a < 0 \,\&\, a \ge b) \,|\, \left((a \,|\, b) = \text{0x80000000}\right)$$

В некоторых приложениях более эффективным может оказаться путь, состоящий в некотором преобразовании чисел и последующем сравнении с плавающей точкой с использованием единственной команды. Например, при сортировке n чисел преобразование придется выполнить для каждого числа только один раз, в то время как количество сравнений составляет, как минимум, $\lceil n \log_2 n \rceil$ (в смысле минимакса).

В табл. 15.2 приведены четыре типа таких преобразований. Для преобразований в левом столбце -0.0 эквивалентно +0.0, в правом — значение $-0.0 < +0.0$. В любом случае смысл сравнения при преобразовании не изменяется. Переменная п — знаковая, t — беззнаковая, а с может быть как знаковой, так и беззнаковой.

Таблица 15.2. Подготовка чисел с плавающей точкой к целым сравнениям

–0.0 = +0.0 (IEEE)	–0.0 < +0.0 (He-IEEE)
`if (n >= 0) n = n+0x80000000;` `else n = -n;` *Используется беззнаковое сравнение*	`if (n >= 0) n = n+0x80000000;` `else n = ~n;` *Используется беззнаковое сравнение*
`c = 0x7FFFFFFF;` `if (n < 0) n = (n ^ c) + 1;` Используется знаковое сравнение	`C = 0x7FFFFFFF;` `if (n < 0) n = n ^c;` Используется знаковое сравнение
`C = 0x80000000;` `if (n < 0) n = c - n;` Используется знаковое сравнение	`c = 0x7FFFFFFF;` `if (n < 0) n = c - n;` Используется знаковое сравнение
`t = n >> 31;` `n = (n ^ (t >>1» - t;` Используется знаковое сравнение	`t = (unsigned) (n>>30) >> 1;` `n = n ^t;` Используется знаковое сравнение

В последней строке приведен код без ветвлений, который может быть реализован на RISC-компьютере с базовым набором команд, с использованием четырех команд в левом столбце, и трех — в правом (эти команды должны быть выполнены для каждого из аргументов операции сравнения).

15.3. Распределение ведущих цифр

Когда IBM выпустила в 1964 году свою ЭВМ System/360, все были озабочены снижением точности арифметики с плавающей точкой. Предыдущая линия ЭВМ, семейство 704–709–7090, имела 36-битовое слово. Представление числа с плавающей точкой с одинарной точностью состояло из 9-битового поля знака и показателя степени, за которым следовали 27 бит двоичного представления дроби. Старший бит дроби явно включался в это представление числа, так что точность представления чисел с плавающей точкой была равна 27 битам.

ЭВМ System/360 имела 32-битовое слово. Для хранения чисел с плавающей точкой одинарной точности IBM был выбран формат с 8-битовым полем знака и показателя степени и 24-битовым полем дробной части. Это снижало точность представления с 27 до 24 бит, что само по себе достаточно плохо, но на самом деле все обстояло еще хуже. Для того чтобы обеспечить большой диапазон показателя степени, каждая единица в 7-битовом показателе приравнивалась к 16. Это привело, по сути, к представлению дробной части в шестнадцатеричном виде, где первая цифра числа может принимать любое двоичное значение от 0001 до 1111 (от 1 до 15). Числа, старшая цифра которых равна 1, таким образом, имели точность всего лишь 21 бит (поскольку три старших бита такого числа нулевые), и таких чисел должно быть примерно 1/15, или 6.7% от всех чисел.

Но на этом беды не кончаются, и на самом деле все обстоит еще хуже. Как тут же было показано теоретическим анализом и эмпирическими экспериментами, ведущие цифры чисел распределены *не* равномерно и среди чисел с плавающей точкой в шестнадцатеричной системе счисления следует ожидать, что единица будет ведущей цифрой у четверти всех чисел.

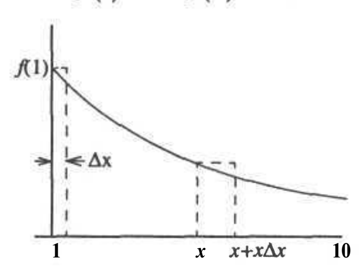
Рассмотрим распределение ведущих цифр у десятичных чисел. Предположим, что у нас есть большое множество чисел, представляющих различные физические величины — длины, объемы, массы и т.п., выраженные в "научной" записи (например, 6.022×10^{23}). Если старшая цифра у таких чисел имеет некоторое распределение, то это распределение не должно зависеть от используемых единиц измерений, например сантиметров или дюймов, фунтов или килограммов и т.д. Таким образом, если умножить все числа данного множества на некоторую константу, распределение старших цифр должно остаться тем же. Например, рассматривая умножение на 2, мы должны сделать вывод о том, что чисел с ведущей единицей (от 1.0 до 1.999..., умноженных на некоторую степень 10) должно быть столько же, сколько и чисел с ведущими цифрами 2 или 3 (от 2.0 до 3.999..., умноженных на некоторую степень 10), поскольку не должно иметь значения, измеряется ли длина в метрах или в полуметрах, масса — в килограммах или полукилограммах и т.п.

Пусть $f(x)$, $1 \le x < 10$, — функция плотности вероятности для ведущих цифр множества чисел с физической размерностью. Эта функция обладает тем свойством, что

$$\int_a^b f(x)\,dx$$

пропорционально количеству чисел, старшая цифра которых находится в диапазоне от *a* до *Ь*. Для малых приращений Ах должно выполняться следующее соотношение (см. приведенный ниже рисунок):

$$f(1)\cdot\Delta x = f(x)\cdot x\Delta x,$$



поскольку $f(1)\cdot\Delta x$ примерно пропорционально количеству чисел в диапазоне от 1 до $1+\Delta x$ (игнорируя множитель степени 10), а $f(x)\cdot x\Delta x$ приблизительно пропорционально количеству чисел в диапазоне от x до $x+x\Delta x$. Так как последнее множество представляет собой первое, умноженное на x, коэффициенты пропорциональности должны быть одинаковы, а следовательно, функция плотности вероятности имеет простейший вид:

$$f(x) = f(1)/x.$$

Поскольку площадь под кривой от $x=1$ до $x=10$ должна быть равна 1 (все числа со старшими цифрами лежат в диапазоне от 1.000... до 9.999...), легко показать, что

$$f(1) = 1/\ln 10.$$

Числа со старшей цифрой в диапазоне от a до b ($1 \le a \le b < 10$) составляют часть общего количества всех чисел, выражаемую формулой

$$\int_a^b \frac{dx}{x\ln 10} = \left.\frac{\ln x}{\ln 10}\right|_a^b = \frac{\ln b/a}{\ln 10} = \log_{10}\frac{b}{a}.$$

Таким образом, в десятичной системе счисления количество чисел со старшей цифрой 1 составляет $\log_{10}(2/1) \approx 30.103\%$ от всех чисел, а чисел со старшей цифрой девять — только $\log_{10}(10/9) \approx 4.58\%$.

В шестнадцатеричной системе счисления доля чисел, старшей цифрой которых является цифра из диапазона $1 \le a \le b < 16$, равна $\log_{16}(b/a)$. Следовательно, в шестнадцатеричной системе счисления доля чисел со старшей цифрой, равной 1, равна $\log_{16}(2/1) = 1/\log_2 16 = 0.25$.

15.4. Таблица различных значений

В табл. 15.3 приведены IEEE-представления различных чисел, имеющие практический интерес. Приведенные значения не являются точными и округлены до ближайшего представимого числа.

Таблица 15.3. IEEE-представления различных значений

Десятичное значение	Одинарная точность (шестнадцатеричное число)	Двойная точность (шестнадцатеричное число)
$-\infty$	FF80 0000	FFFO 0000 0000 0000
-2.0	C000 0000	C000 0000 0000 0000
-1.0	BF80 0000	BFFO 0000 0000 0000
–0.5	BFOO 0000	BFEO 0000 0000 0000
–0.0	8000 0000	8000 0000 0000 0000
+0.0	0000 0000	0000 0000 0000 0000
Наименьшее положительное денормализованное	0000 0001	0000 0000 0000 0001
Наибольшее денормализованное	007F FFFF	OOOF FFFF FFFF FFFF
Наименьшее положительное нормализованное	0080 0000	0010 0000 0000 0000
$\pi/180$ (0.01745...)	3C8E FA35	3F91 BF46 A252 9D39
0.1	3DCC CCCD	3FB9 9999 9999 999A
$\log_{10} 2$ (0.3010...)	3E9A 209B	3FD3 4413 509F 79FF
$1/e$ (0.3678...)	3EBC 5AB2	3FD7 8B56 362C EF38
$1/\ln 10$ (0.4342...)	3EDE 5BD9	3FBB CB7B 1526 E50E
0.5	3FOO 0000	3FEO 0000 0000 0000
$\ln 2$ (0.6931...)	3F31 7218	3FE6 2E42 FEFA 39EF
$1/\sqrt{2}$ (0.7071...)	3F35 04F3	3FE6 A09E 667F 3BCD
1/1п3(0.9102...)	3F69 0570	3FED 20AE O3BC C153
1.0	3F80 0000	3FFO 0000 0000 0000
1п3 (1.0986...)	3F8C 9F54	3FF1 93EA 7AAD 030B
$\sqrt{2}$ (1.414...)	3FB5 04F3	3FF6 A09E 667F 3BCD
$1/\ln 2$ (1.442...)	3FB8 AA3B	3FF7 1547 652B 82FE
$\sqrt{3}$ (1.732...)	3FDD B3D7	3FFB B67A E858 4CAA

Окончание табл. 15.3

Десятичное значение	Одинарная точность (**шестнадцатеричное** число)	Двойная точность (**шестнадцатеричное** число)
2.0	4000 0000	4000 0000 0000 0000
ln10 (2.302...)	4013 5D8E	4002 6BB1 BBB5 5516
е (2.718...)	402D F854	4005 BFOA 8B14 5769
3.0	4040 0000	4008 0000 0000 0000
тс (3.141...)	4049 OFDB	4009 21FB 5444 2D18
$\sqrt{10}$ (3.162...)	404A 62C2	4009 4C58 3ADA 5B53
$\log_2 10$ (3.321...)	4054 9A78	400A 934F 0979 A371
4.0	4080 0000	4010 0000 0000 0000
5.0	40AO 0000	4014 0000 0000 0000
6.0	40CO 0000	4018 0000 0000 0000
2π (6.283...)	40C9 OFDB	4019 21FB 5444 2D18
7.0	40EO 0000	401C 0000 0000 0000
8.0	4100 0000	4020 0000 0000 0000
9.0	4110 0000	4022 0000 0000 0000
10.0	4120 0000	4024 0000 0000 0000
11.0	4130 0000	4026 0000 0000 0000
12.0	4140 0000	4028 0000 0000 0000
13.0	4150 0000	402A 0000 0000 0000
14.0	4160 0000	402C 0000 0000 0000
15.0	4170 0000	402E 0000 0000 0000
16.0	4180 0000	4030 0000 0000 0000
$180/\pi$ (57.295...)	4265 2EE1	404C A5DC 1A63 C1F8
$2^{23}-1$	4AFF FFFE	415F FFFF COOO 0000
2^{23}	4BOO 0000	4160 0000 0000 0000
$2^{24}-1$	4B7F FFFF	416F FFFF EOOO 0000
2"	4B80 0000	4170 0000 0000 0000
$2^{31}-1$	4FOO 0000	41DF FFFF FFCO 0000
2^{31}	4FOO 0000	41EO 0000 0000 0000
$2^{32}-1$	4F80 0000	41EF FFFF FFEO 0000
2^{32}	4F80 0000	41FO 0000 0000 0000
2^{52}	5980 0000	4330 0000 0000 0000
2^{63}	5FOO 0000	43EO 0000 0000 0000
2^{64}	5F80 0000	43FO 0000 0000 0000
Наибольшее нормализованное	7F7F FFFF	7FEF FFFF FFFF FFFF
oo	7F80 0000	7FFO 0000 0000 0000
"Наименьшее нечисло" SNaN	7F80 0001	7FFO 0000 0000 0001
"Наибольшее нечисло" SNaN	7FBF FFFF	7FF7 FFFF FFFF FFFF
"Наименьшее нечисло" QNaN	7FCO 0000	7FF8 0000 0000 0000
"Наибольшее нечисло" QNaN	7FFF FFFF	7FFF FFFF FFFF FFFF

Стандарт IEEE 754 не оговаривает, каким образом различаются сигнализирующие нечисла (SNaN) и несигнализирующие нечисла (QNaN). В табл. 15.3 использовано соглашение, применяющееся на платформах PowerPC, AMD 29050, Intel x86 и I860: старший значащий бит дробной части равен 0 для SNaN и 1 — для QNaN. На платформах Compaq Alpha, HP PA-RISC и MIPS используется тот же бит, но с обратным значением (O — QNaN, 1 — SNaN).

ГЛАВА 16

ФОРМУЛЫ ДЛЯ ПРОСТЫХ ЧИСЕЛ

16.1. Введение

Как и все молодые студенты, я в свое время был очарован простыми числами и пытался найти формулу для их поиска. Я не знал точно, какие операции должны использоваться в этой "формуле", и даже не очень представлял себе, какую именно функцию я ищу: функцию, которая по заданному n будет возвращать n-е простое число, или по заданному простому числу будет находить следующее простое число, или будет просто давать простые числа, хотя и не все из них. Сейчас, наперекор всем этим неоднозначностям, я хочу рассмотреть, что же все же известно нам по этой проблеме? В результате вы узнаете, что *формулы* для простых чисел существуют, но все они не очень удовлетворительны.

Начнем наше рассмотрение с краткой истории вопроса.

В 1640 году Ферма предположил, что формула

$$F_n = 2^{2^n} + 1$$

всегда дает простые числа, и числа этого вида получили название "чисел Ферма". Предположение Ферма справедливо для и от 0 до 4, но в 1732 году Эйлер обнаружил, что

$$F_5 = 2^{2^5} + 1 = 641 \cdot 6\,700\,417 .$$

(Мы уже имели дело с этими множителями, когда рассматривали вопросы деления на константу на 32-битовом компьютере.) Затем в 1880 году Ландри показал, что

$$F_6 = 2^{2^6} + 1 = 274\,177 \cdot 67\,280\,421\,310\,721 .$$

В настоящее время известно, что F_n — составные числа для многих больших значений и, в частности для и от 7 до 16 включительно. В настоящее время неизвестно ни одно значение $n > 4$, для которого число Ферма было бы простым [33]. Увы, гипотеза оказалась слишком поспешной...[1]

Почему же Ферма использовал двойное возведение в степень? Он знал, что если m имеет нечетный множитель, отличный от 1, то 2" +1 — составное число. Если $m = ab$, где b нечетно и не равно 1, то

$$2^{ab} + 1 = \left(2^a + 1\right)\left(2^{a(b-1)} - 2^{a(b-2)} + 2^{a(b-3)} - \ldots + 1\right).$$

Зная это, Ферма заинтересовался числами $2^m + 1$, такими, где m не имеет ни одного нечетного множителя, отличного от 1, т.е. $m = 2^n$. Он испробовал несколько значений n и убедился, что они дают простые числа $2^{2^n} + 1$.

Практически все согласны, что полином вполне можно считать "формулой". В 1772 году Эйлером был открыт удивительный полином

$$f(n) = n^2 + n + 41 ,$$

---

[1] Справедливости ради надо заметить, что это единственная известная ошибочная гипотеза Ферма [61].

обладающий тем свойством, что он дает простые числа для всех я от 0 до 39. Этот результат может быть расширен. Так как

$$f(-n)=n^2-n+41=/(\text{л}-1),$$

$f(-n)$ дает простые числа для каждого n из диапазона от 1 до 40, т.е. $f(n)$ дает простые числа для всех п от -1 до –40. Таким образом, полином

$$f(n-40)=(n-40)^2+(n-40)+41=n^2-79n+1601$$

дает простые числа для всех n от 0 до 79. (Однако эстетическая привлекательность формулы при этом теряется, так как она дает повторяющиеся и немонотонные результаты — для $n=0, 1 \ldots 79$ формула дает числа 1601, 1523, 1447, ..., 43, 41, 41, 43, ..., 1447, 1523, 1601.)

Несмотря на всю привлекательность полинома, открытого Эйлером, известно, что не существует полиномиальной формулы $f(n)$, которая давала бы простое число для каждого и (не считая тривиального случая константного полинома типа $f(n)=5$). Более того, справедлива следующая теорема [33].

Теорема. *Если* $f(n)=P(n,2^n,3^n,\ldots,k^n)$ *представляет собой полином с целочисленными коэффициентами от своих аргументов и* $f(n)\to\infty$ *при* $n\to\infty$, *то* $f(n)$ *является составным значением для бесконечного множества значений n.*

Следовательно, формула типа $n^2\cdot 2^n+2n^3+2n+5$ должна давать бесконечное количество составных чисел. С другой стороны, теорема ничего не говорит о формулах, содержащих члены наподобие 2^{2^n}, л" или $n!$.

Формула для n-го целого простого числа может быть получена при помощи функции получения максимального целого, не превосходящего данное число ("пол"), и магического значения

$$a=0.203005000700011000013\ldots$$

Число a в десятичной системе счисления представляет собой первое простое число, записанное в первой позиции после десятичной точки, второе простое число, записанное в следующих двух позициях, третье, записанное в следующих трех позициях и т.д. При этом для очередного простого числа всегда найдется достаточно свободного места, так как $p_n<10^n$. Не доказывая этого, просто заметим: поскольку известно, что между и и 2л $(n\ge 2)$ всегда есть простое число, значит, между n и $10n$ оно есть наверняка, а отсюда следует, что $p_n<10^n$. Формула для n-го простого числа выглядит следующим образом:

$$p_n=\left\lfloor 10^{\frac{n^2+n}{2}}a\right\rfloor-10^n\left\lfloor 10^{\frac{n^2-n}{2}}a\right\rfloor,$$

где было использовано соотношение

$$\sum_{i=1}^{n}\text{л}=\frac{n^2+n}{2}.$$

Например:

$$p_3=\lfloor 10^6a\rfloor-10^3\lfloor 10^3a\rfloor=203005-203000=5.$$

Это довольно дешевый трюк, который требует знания окончательного результата для корректного определения a. Формула такого рода может представлять интерес, только если имеется некоторый путь определения a независимо от знания всех простых чисел, но, увы, никто такого способа определения a пока что не знает.

Понятно, что такая формула может служить для многих различных последовательностей, но сейчас нас это не интересует.

16.2. Формулы Вилланса

Первая формула

Виллане (C.P. Willans) дал следующую формулу для **n-го** простого числа **[62]:**

$$p_n = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{n} \left(\sum_{x=1}^{m} \left\lfloor \cos^2 \pi \frac{(x-1)!+1}{x} \right\rfloor \right)^{-1/n} \right\rfloor .$$

Вывод этой формулы начинается с теоремы Вильсона, которая гласит, что p является простым или равно 1 тогда и только тогда, когда $(p-1)! \equiv -1 \pmod{p}$. Таким образом,

$$\frac{(x-1)!+1}{x}$$

будет целым для простого x (или A= 1) и дробным для всех составных x. Следовательно,

$$F(x) = \left\lfloor \cos^2 \pi \frac{(x-1)!+1}{x} \right\rfloor = \begin{cases} 1, & x \text{ простое или } 1, \\ 0, & x \text{ составное.} \end{cases} \qquad (1)$$

Таким образом, если $\pi(m)$ **обозначает**[2] количество простых чисел, не превышающих m, то

$$\pi(m) = -1 + \sum_{x=1}^{m} F(x) . \qquad (2)$$

Заметим, что $\pi(p_n) = n$, а кроме того,

$$\pi(m) < n \text{ для } m < p_n \text{ и}$$

$$\pi(m) \ge n \text{ для } m \ge p_n .$$

Таким образом, количество значений m от 1 до ∞, для которых $\pi(m) < n$, равно $p_n - 1$, т.е.

$$p_n = 1 + \sum_{m=1}^{\infty} (\pi(m) < n) , \qquad (3)$$

где под знаком суммы стоит выражение-предикат, значение которого может быть равно О или 1.

Поскольку у нас есть формула для $\pi(m)$, уравнение (3) представляет собой формулу **для n-го** простого числа как функцию n. Однако в ней есть две вещи, которые делают ее неприемлемой: бесконечная сумма и использование выражения-предиката, которое обычно в математике не применяется.

---

[2] Автор приносит извинения за разное по смыслу применение символа κ в столь тесной близости одно к другому, но это стандартное обозначение, использование которого не должно привести к каким-либо трудностям при чтении книги.

Доказано, что для $n \geq 1$ имеется, как минимум, одно простое число между n и $2n$. Следовательно, количество простых чисел, не превышающих 2^n, по меньшей мере равно n, т.е. $\pi(2^n) \geq n$. Таким образом, предикат $\pi(m) < n$ для $m \geq 2^n$ равен 0, так что верхний предел суммирования можно заменить на 2^n.

Виллане использовал более искусную замену для выражения-предиката. Пусть

$$\mathrm{LT}(x, y) = \left\lfloor \sqrt[y]{\frac{y}{1+x}} \right\rfloor \text{ для } x = 0,1,2\ldots;\ y = 1,2,\ldots$$

Тогда, если $x < y$, то $1 \leq y/(1+x) \leq y$, так что $1 \leq \sqrt[y]{y/(1+x)} \leq \sqrt[y]{y} < 2$ Кроме того, если $x \geq y$, то $0 < y/(1+x) < 1$, так что $0 < \sqrt[y]{y/(1+x)} < 1$. Применяя функцию наибольшего целого, не превосходящего заданное число ("пол"), получим

$$\mathrm{LT}(x, y) = \begin{cases} 1, & x < y, \\ 0, & x \geq y, \end{cases}$$

так что $\mathrm{LT}(x, y)$ представляет собой предикат $x < y$ (для x и y из указанных диапазонов).

Подставив полученное выражение в (3), получим

$$p_n = 1 + \sum_{m=1}^{2^n} \mathrm{LT}(\pi(m), n) = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{\frac{n}{1+\pi(m)}} \right\rfloor.$$

Дальнейшая подстановка уравнения (2) для выражения $\pi(m)$ через $F(x)$ и уравнения (1) для $F(x)$ дает нам формулу, приведенную в начале этого раздела.

Вторая формула

Виллане дал еще одну формулу для n-го простого числа:

$$p_n = \sum_{m=1}^{2^n} m F(m) \left\lfloor 2^{-|\pi(m)-n|} \right\rfloor.$$

Здесь F и π — функции, использовавшиеся в первой формуле. Таким образом, $mF(m)$ равно m, если m — простое число или единица, и 0, если m — составное число. Третий множитель представляет собой предикат $\pi(m) = n$. Все члены суммы равны О, кроме одного, равного n-му простому числу. Например:

$$p_4 = 1\cdot 1\cdot 0 + 2\cdot 1\cdot 0 + 3\cdot 1\cdot 0 + 4\cdot 0\cdot 0 + 5\cdot 1\cdot 0 + 6\cdot 0\cdot 0 + 7\cdot 1\cdot 1 +$$
$$+ 8\cdot 0\cdot 1 + 9\cdot 0\cdot 1 + 10\cdot 0\cdot 1 + 11\cdot 1\cdot 0 + \ldots + 16\cdot 0\cdot 0 = 7.$$

Третья формула

На этом Виллане не остановился и предложил еще одну формулу для n-го простого числа, которая не использует ни одной "неаналитической"[3] функции типа "пол" или абсолютное значение величины. Он заметил, что для $x = 2, 3, \ldots$, функция

[3] Это термин автора книги, а не Вилланса.

$$\frac{((x-1)!)^2}{x} = \begin{cases} \text{целое} + \frac{1}{x}, & \text{если } x \text{ - простое число,} \\ \text{целое}, & \text{если } x \text{ - составное число или } 1. \end{cases}$$

Первая часть следует из того, что

$$\frac{((x-1)!)^2}{x} = \frac{((x-1)!+1)\cdot((x-1)!-1)}{x} + \frac{1}{x}$$

и $(x-1)!+1$ делится на x в соответствии с теоремой Вильсона. Таким образом, предикат "x — простое число" для $x > 2$ задается следующим образом:

$$H(x) = \frac{\sin^2 \pi \frac{((x-1)!)^2}{x}}{\sin^2 \frac{\pi}{x}}.$$

Из этого следует

$$\pi(m) = \sum_{x=2}^{m} H(x), \ m = 2,3,\ldots$$

Эту формулу нельзя преобразовать в формулу для p_n методами, использованными в первых двух формулах Вилланса, так как в них применяется функция "пол". Вместо этого Виллане предложил следующую формулу[4] для предиката $x < y$ при $x, y \geq 1$:

$$\mathrm{LT}(x, y) = \sin\left(\frac{\pi}{2}\cdot 2^e\right),$$

где

$$e = \prod_{i=0}^{y-1}(x-i).$$

Таким образом, если $x<y$, то $e = x(x-1)\ldots(0)(-1)\ldots(x-(y-1)) = 0$, так что $\mathrm{LT}(x, y) = \sin(\pi/2) = 1$. Если $x \geq y$, произведение не включает 0, так что $e \geq 1$ и $\mathrm{LT}(x,y) = \sin((\pi/2)\cdot(\text{четное число})) = 0$.

Наконец, как и в первой формуле Вилланса,

$$p_n = 2 + \sum_{m=2}^{2^n} \mathrm{LT}(\pi(m), n).$$

Объединив все вместе, получим следующую "жуть":

$$p_n = 2 + \sum_{m=2}^{2^n} \sin\left(\frac{\pi}{2}\cdot 2^{\prod_{i=0}^{n-1}\left(\sum_{x=2}^{m}\frac{\sin^2 \pi\frac{((x-1)!)^2}{x}}{\sin^2\frac{\pi}{x}} - i\right)}\right).$$

[4] Мы немного упростили эту формулу.

Четвертая формула

После этого Виллане приводит еще одну формулу, которая выражает простое число p_{n+1} через p_n:

$$p_{n+1} = 1 + p_n + \sum_{i=1}^{2p_n} \prod_{j=1}^{i} f(p_n + j),$$

где $f(x)$ — предикат "x — составное число", $x \geq 2$, т.е.

$$f(x) = \left\lfloor \cos^2 \pi \frac{((x-1)!)^2}{x} \right\rfloor.$$

В качестве альтернативы можно использовать соотношение $/(x) = 1 - H(x)$, что позволит избежать применения функции "пол".

Рассмотрим пример работы этой формулы для $p_n = 7$. Тогда

$$\begin{aligned} p_{n+1} &= 1 + 7 + f(8) + f(8)f(9) + f(8)f(9)f(10) + \\ &+ f(8)f(9)f(10)f(11) + \ldots + f(8)f(9)\ldots f(14) = \\ &= 1 + 7 + 1 + 1 \cdot 1 + 1 \cdot 1 \cdot 1 + 1 \cdot 1 \cdot 1 \cdot 0 + \ldots 1 \cdot 1 \cdot 1 \cdot 0 \cdot 1 \cdot 0 \cdot 1 = 11. \end{aligned}$$

16.3. Формула Вормелла

Вормелл (C.P. Wormell) [63] усовершенствовал формулу Вилланса, убрав из нее как тригонометрические функции, так и функцию "пол". Вычисления по формуле Вормелла в принципе могут быть проведены с помощью простой компьютерной программы с использованием только целочисленной арифметики. Ее вывод не использует теорему Вильсона. Вормелл начал с того, что для $x \geq 2$

$$B(x) = \prod_{a=2}^{x} \prod_{b=2}^{x} (x - ab)^2 = \begin{cases} \text{положительное целое, если } x \text{ - простое число,} \\ 0, \text{ если } x \text{ - составное число.} \end{cases}$$

Таким образом, количество простых чисел, не превосходящих m, задается соотношением

$$\pi(m) = \sum_{x=2}^{m} \frac{1 + (-1)^{2^{B(x)}}}{2},$$

потому что под знаком суммы находится предикат "x — простое число".

Заметим, что для $n > 1$, $a > O$

$$\prod_{r=1}^{n} (1 - r + a)^2 = \begin{cases} 0, \text{ если } a < n, \\ \text{положительное целое, если } a \geq n. \end{cases}$$

Повторяя описанный выше прием, находим, что предикат $a < n$ можно выразить следующим образом:

$$(a < n) = \frac{1 - (-1)^{2^{\prod_{r=1}^{n}(1-r+a)^2}}}{2}.$$

Поскольку

$$p_n = 2 + \sum_{m=2}^{2^n} \left(\pi(m) < n\right),$$

после вынесения постоянных множителей за знак суммы получаем

$$p_n = \frac{3}{2} + 2^{n-1} - \frac{1}{2} \sum_{m=2}^{2^n} (-1)^{2^{\prod_{r=1}^{n} \left(1 - r + \frac{(m-1)}{2} + \frac{1}{2} \sum_{x=2}^{m} (-1)^{2^{\prod_{a=2} \prod_{b=2} (x-ab)^2}}\right)^2}}.$$

Как и было обещано, формула **Вормелла** не использует тригонометрических функций. Однако, как указал Вормелл, они появятся вновь, если заменить степени -1, используя соотношение (-1)" $= \cos \pi n$.

16.4. Формулы для других сложных функций

Еще раз внимательно посмотрим на то, что сделали Виллане и Вормелл. Ниже постулированы правила, определяющие, что именно мы подразумеваем под классом функций, которые могут быть представлены "формулами" и которые мы назовем "формульными функциями". Здесь Г означает сокращение $x_1, x_2, \ldots, x_n$ для любого $n \geq 1$. Область значений представляет собой целые числа ... -2, $-1, 0, 1, 2, \ldots$.

Константы $\ldots -1, 0, 1, \ldots$ являются формульными функциями.

Функции отображения $f(\bar{x}) = x_i$ для $1 < i < n$ являются формульными.

Выражения $x + y$, $x - y$ и xy являются формульными функциями, если таковыми являются x и y.

Класс формульных функций замкнут по отношению к суперпозиции функций (подстановке), т.е. $f\left(g_1(\bar{x}), g_2(\bar{x}), \ldots, g_m(\bar{x})\right)$ является формульной функцией, если/ и g_i являются таковыми для $i = 1, \ldots, m$.

Ограниченные суммы и произведения

$$\sum_{i=a(\bar{x})}^{b(\bar{x})}{}' f(i, \bar{x}) \text{ и } \prod_{i=a(\bar{x})}^{b(\bar{x})}{}' f(i, \bar{x})$$

представляют собой формульные функции, если a, b и / являются таковыми и $a(\bar{x}) \leq b(\bar{x})$.

Требование ограниченности сумм и произведений сохраняет вычислительный характер формул, т.е. формулы могут быть вычислены подстановкой значений аргументов и проведением конечного числа операций. Причина наличия штриха у символов £ и П поясняется далее в этой главе.

При формировании новых формульных функций с использованием суперпозиции в случае необходимости применяются круглые скобки в соответствии со стандартными соглашениями об их применении.

Заметим, что деление в приведенный выше список не включено. Однако даже при этом приведенный список нельзя считать минимальным. Конечно, интересно найти минимально необходимый список правил, но не будем останавливаться на этом вопросе в нашей книге.

Определение "формульной функции" близко к определению "элементарной функции", приведенному в [9]. Однако область значений, использованная в [9], представляет собой неотрицательные целые числа (как и в обычной теории рекурсивных функций). Кроме того, в [9] требуется, чтобы границы итерационных сумм и произведений были О и $x-1$ (где x — переменная) и допускали пустые диапазоны (в этом случае сумма считается равной 0, а произведение — 1).

Далее покажем, что класс формульных функций существенно шире и включает большинство функций, встречающихся в математике. Тем не менее он не включает в себя все функции, которые легко определяются и имеют элементарный характер.

По сравнению с теорией рекурсивных функций наши выводы несколько усложнены, так как здесь переменные могут иметь отрицательные значения. Однако то, что некоторые значения могут быть отрицательными, легко исправить, используя возведение соответствующего выражения в квадрат. Еще одним усложнением является настойчивое требование. чтобы диапазоны сумм и произведений были непустыми.

В нашем рассмотрении под "предикатом" понимается функция, которая возвращает значение, равное 0 или 1, в то время как в теории рекурсивных функций предикат представляет собой функцию, которая возвращает значение истинно/ложно, и каждый предикат имеет связанную с ним "характеристическую функцию", которая возвращает значения 0/1. Мы ассоциируем значение 1 с истиной, а 0 с ложью, как обычно делается в языках программирования и в вычислительных машинах в целом (в плане работы команд *и* и *или*). Однако в теории логических и рекурсивных функций встречается и противоположная ассоциация логических и числовых значений.

Приведем ряд примеров формульных функций.

1. $a^2 = aa$, $a^3 = aaa$ и т.д.

2. Предикат $a = b$:

$$(a=b)=\prod_{j=0}^{(a-b)^2}(1-j).$$

3. $(a \neq b) = 1-(a-b)$.

4. Предикат $a > b$:

$$(a \geq b)=\sum_{i=0}^{(a-b)^2}((a-b)=i)=\sum_{i=0}^{(a-b)^2}\prod_{j=0}^{((a-b)-i)^2}(1-j).$$

Теперь можно объяснить, почему не используется соглашение, в соответствии с которым итеративная сумма/произведение принимают при пустом диапазоне суммирования/произведения значения 0/1. Если поступить подобным образом, то можно получить такие "жульничества", как

$$(a=b)=\sum_{i=0}^{-(a-b)^2}1 \text{ и } (a \geq b)=\prod_{i=a}^{b-1}0.$$

Предикаты сравнений являются ключом ко всему последующему материалу, поэтому не желательно, чтобы они были основаны на искусственных построениях такого рода.

5. $(a > b) = (a \geq b+1)$.

6. $(a \leq b) = (b \geq a)$.

7. $(a<b) = (b > a)$.

8. $|в| = (2(a \geq 0)-1)a$.

9. $\max(a,b) = (a \geq b)(a-b)+b$.

10. $\min(a,b) = (a \geq b)(b-a)+a$.

Теперь можно скорректировать итерационные суммы и произведения таким образом, чтобы они давали корректный результат и при пустом диапазоне суммирования или произведения.

11. $\sum_{i=a(\overline{x})}^{b(\overline{x})} f(i,\overline{x}) = (b(\overline{x}) \geq a(\overline{x})) \sum_{i=a(\overline{x})}^{\max(a(\overline{x}),b(\overline{x}))}{}' f(i,\overline{x})$.

12. $\prod_{i=a(\overline{x})}^{b(\overline{x})} f(i,\overline{x}) = 1+\left(b(\overline{x}) \geq a(\overline{x})\right)\left(-1+\prod_{i=a(\overline{x})}^{\max(a(\overline{x}),b(\overline{x}))}{}' f(i,\overline{x})\right)$.

Начиная с этого момента, символы суммы и произведения используются без штриха. Таким образом, все дальнейшие определения функций корректны для любых значений аргументов.

13. $n! = \prod_{i=1}^{n} i$. Это определение дает нам $n! = 1$ для $n \leq 0$.

В формулах ниже P и Q обозначают предикаты.

14. $\neg P(\overline{x}) = 1-P(\overline{x})$.

15. $P(\overline{x}) \& Q(\overline{x}) = P(\overline{x})Q(\overline{x})$.

16. $P(\overline{x}) | Q(\overline{x}) = 1-(1-P(\overline{x}))(1-Q(\overline{x}))$.

17. $P(\overline{x}) \oplus Q(\overline{x}) = (P(\overline{x})-Q(\overline{x}))^2$.

18. if $P(\overline{x})$ then $f(\overline{y})$ else $g(\overline{z}) = P(\overline{x})f(\overline{y}) + (1-P(\overline{x}))g(\overline{z})$.

19. a" = if $n \geq 0$ then $\prod_{i=1}^{n} a$ else 0.

Эта формула дает результат 0 для ж 0 и 1 для 0^0, что в ряде случаев может оказаться некорректным решением.

20. $(m \leq \forall x \leq n) P(x,\overline{y}) = \prod_{x=m}^{n} P(x,\overline{y})$.

21. $(m \leq \exists x \leq n) P(x,\overline{y}) = 1-\prod_{x=m}^{n}(1-P(x,\overline{y}))$.

Пустое V истинно, пустое ∃ ложно.

22. $(m \le \min x \le n) P(x, \bar{y}) = m + \sum_{i=m}^{n} \prod_{j=m}^{i} \left(1 - P(j, \bar{y})\right)$.

Значение этого выражения есть наименьшее x в диапазоне от m до л, такое, что предикат для него оказывается истинным, либо m в случае пустого **диапазона**, либо и $+ 1$, если предикат ложен для всего (непустого) **диапазона. Эта** операция называется "ограниченной минимизацией" и **является** весьма мощным инструментом при разработке **но**вых формульных функций. Вычисление такой минимизации, представляющей собой тип функциональной инверсии, предложено Гудштейном (Goodstein) [23].

23. $\lfloor \sqrt{n} \rfloor = (0 \le \min k \le |n|)\left((k+1)^2 > n\right)$.

Эта функция представляет собой "целочисленный квадратный корень", который для обобщенности равен 0 при ж 0 .

24. $d \mid n = (-|n| \le \exists q \le |n|)(n = qd)$.

Это предикат "*d***является** делителем п", в соответствии с которым $0 \mid 0$, но $\neg(0 \mid n)$ при $n \ne 0$.

25. $n \div d =$ if $n \ge 0$ then $(-n \le \min q \le n)(0 \le \exists r \le |d| - 1)(n = qd + r)$
else (л $< \min q \le -n)(-|d| + 1 \le \exists r \le 0)(n = qd + r)$.

Это формула для отсекающего целочисленного деления. Для $d = 0$ формула произвольно дает результат $|n| + 1$.

26. $\mathrm{rem}(n, d) = n - (n \div d)d$.

Это обычная функция получения остатка при делении. Если $\mathrm{rem}(n, d)$ имеет ненулевое значение, его знак равен знаку числителя *п*. Если d - 0 , остаток равен я.

27. $\mathrm{isprime}(n) = n \ge 2 \mathbin{\&} \neg(2 \le \exists d \le |n| - 1)(d \mid n)$.

28. $\pi(n) = \sum_{i=1}^{n} \mathrm{isprime}(i)$ (количество простых чисел, не превосходящих и).

29. $p_n = (1 \le \min k \le 2^n)(\pi(k) = n)$.

30. $\mathrm{exponent}(p, n) = (0 \le \min x \le |n|) \neg (p^{x+1} \mid n)$.

Это формула показателя степени простого множителя p данного числа п > 1 .

31. Для $n \ge 0$:

$2^n = \prod_{i=1}^{n} 2$, $2^{2^n} = \prod_{i=1}^{2^n} 2$, $2^{2^{2^n}} = \prod_{i=1}^{2^{2^n}} 2$ и т.д.

32. ***n*-я** цифра десятичного представления $\sqrt{2}$:

$\text{rem}\left(\left\lfloor \sqrt{2\cdot 10^{2n}} \right\rfloor, 10\right)$.

Таким образом, класс формульных функций весьма большой. Тем не менее он ограничен, как минимум, следующей теоремой.

Теорема. *Если f — формульная функция, то существует константа k, такая, что*

$$f(\overline{x}) \le 2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}},$$

где количество двоек равно k.

Эту теорему можно доказать, показав, что каждое из правил 1-5 сохраняет справедливость теоремы. Например, если $f(\overline{x}) = c$ (правило 1), то для некоторого h

$$f(\overline{x}) \le \left.2^{2^{\cdot^{\cdot^{2}}}}\right\}h,$$

где имеется h двоек. Таким образом,

$$f(\overline{x}) \le \left.2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\right\}h + 2,$$

потому что $\max(|x_1|,\ldots,|x_n|) \ge 0$.

Для $f(\overline{x}) = x_i$ (правило 2) $f(\overline{x}) \le \max(|x_1|,\ldots,|x_n|)$, так что теорема выполняется с $k = 0$.

Рассмотрим правило 3. Пусть

$$f(\overline{x}) \le \left.2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\right\}k_1 \text{ и } g(\overline{x}) \le \left.2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\right\}k_2 \le.$$

Тогда очевидно, что

$$f(\overline{x}) \pm g(\overline{x}) \le 2\cdot \left.2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\right\}\max(k_1,k_2) \le$$

$$\le \left.2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\right\}\max(k_1,k_2)+1.$$

Аналогично можно показать, что теорема выполняется для $f(x,y) = xy$.

Доказательства того, что правила 4 и 5 сохраняют справедливость теоремы, утомительны и сложны, так что здесь они не приводятся.

Из теоремы следует, что функция

$$f(x) = \left.2^{2^{\cdot^{\cdot^{2^{x}}}}}\right\}x \tag{4}$$

не является формульной, поскольку всегда существует достаточно большое x, для которого значение из (4) превышает значение выражения с любым фиксированным количеством двоек k.

Для тех, кто интересуется теорией рекурсивных функций, укажем, что (4) представляет собой примитивную рекурсию. Кроме того, можно легко показать непосредственно из определения примитивной рекурсии, что формульные функции представляют собой примитивную рекурсию. Таким образом, класс формульных функций представляет собой

истинное подмножество примитивных рекурсивных функций. Заинтересовавшемуся этим вопросом читателю можно посоветовать обратиться к [9].

В данной главе представлена не только формула для n-го простого числа, состоящая из элементарных функций, но и многие другие функции, встречающиеся в математике. Кроме того, наши "формульные функции" не используют тригонометрические функции, функцию "пол", абсолютное значение, степени -1 и даже деление. Они очень ловко используют тот факт, что произведение большого количества чисел равно 0, если хотя бы одно из них равно 0 (см. формулу для предиката $a = b$).

Впрочем, после знакомства с приведенными функциями для простых чисел они перестают казаться столь интересными, и вопрос об "интересных" формулах для простых чисел остается открытым. Например, в [53] приведена удивительная теорема Миллса (W.H. Mills) о том, что существует 9, такое, что выражение

$$\left\lfloor \theta^{3^n} \right\rfloor$$

дает простые числа для всех n £ 1. На самом деле имеется бесконечное число таких значений (например, 1.3063778838+ и 1.4537508625483+). Кроме того, нет ничего выдающегося и в числе 3: теорема остается верна, если заменить тройку любым большим ее целым числом (разумеется, для других значений 0). Более того, тройку можно заменить даже двойкой, если справедливо не доказанное до сих пор утверждение, что между n^- и $(и + 1)^2$ всегда имеется по крайней мере одно простое число. Более того... Впрочем, заинтересовавшийся читатель сам найдет в [13, 53] множество формул такого типа.

ПРИЛОЖЕНИЕ А

АРИФМЕТИЧЕСКИЕ ТАБЛИЦЫ ДЛЯ 4-БИТОВОЙ МАШИНЫ

Во всех таблицах данного приложения подчеркивание обозначает знаковое переполнение.

Таблица АЛ. Сложение

									-8	-7	-6	-5	-4	-3	-2	-1
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	£	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	2	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	2	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	2	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	2	A	B	C	D	E	F	10	11	12	13	14	15	16
-8 8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
-7 9	9	A	B	C	D	E	F	10	11	12	11	14	15	16	17	18
-6 A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
-5 B	B	C	D	E	F	10	11	12	11	14	15	16	17	18	19	1A
-4 C	C	D	E	F	10	11	12	13	11	15	li	17	18	19	1A	1B
-3 D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
-2 E	E	F	10	11	12	13	14	15	li	17	18	19	1A	1B	1C	1D
-1 F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

В таблице для вычитания (табл. А.2) предполагается, что бит переноса для a -Ъ устанавливается как и для $a + \bar{Б} + 1$, так что перенос эквивалентен отсутствию заема.

Таблица А.2. Вычитание (строка – столбец)

									-8	-7	-6	-5	-4	-3	-2	-1
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1
1	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2
2	12	11	10	F	E	D	C	B	A	2	8	7	6	5	4	3
3	13	12	11	10	F	E	D	C	B	A	2	8	7	6	5	4
4	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5
5	15	14	13	12	11	10	F	E	D	C	B	A	2	8	7	6
6	16	15	14	13	12	11	10	F	E	D	C	B	A	2	8	7
7	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	в
-8 8	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9
-7 9	19	18	17	16	15	14	11	12	11	10	F	E	D	C	B	A
-6 A	1A	19	18	11	li	15	11	13	12	11	10	F	E	D	C	B
-5 B	1B	1A	19	18	17	li	15	14	13	12	11	10	F	E	D	C
-4 C	1C	1B	1A	19	18	17	li	15	14	13	12	11	10	F	E	D
-3 D	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E
-2 E	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F
-1 F	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10

В случае умножения (табл. А.3 и А.4) переполнение означает, что результат не может быть выражен 4-битовой величиной. В случае знакового умножения (см. табл. А.3) это эквивалентно тому, что первые 5 битов 8-битового результата не равны пяти единицам или пяти нулям.

Таблица А.3. Знаковое умножение

		0	1	2	3	4	5	6	7	-8 8	-7 9	-6 A	-5 B	-4 C	-3 D	-2 E	-1 F
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	2	3	4	5	6	7	F8	F9	FA	FB	FC	FD	FE	FF
	2	0	2	4	6	8	A	C	E	F0	F2	F4	F6	F8	FA	FC	FE
	3	0	3	6	9	C	F	12	15	E8	EB	EE	F1	F4	F7	FA	FD
	4	0	4	8	C	10	14	18	1C	E0	E4	E8	EC	F0	F4	F8	FC
	5	0	5	A	F	14	19	**1E**	23	D8	DD	E2	E7	EC	F1	F6	FB
	6	0	6	C	12	18	**1E**	24	2A	D0	D6	DC	E2	E8	EE	F4	FA
	7	0	7	E	15	1C	23	2A	31	C8	CF	D6	DD	E4	EB	F2	F9
-8	8	0	F8	F0	E8	E0	D8	D0	C8	40	38	30	28	20	18	10	8
-7	9	0	F9	F2	EB	E4	DD	D6	CF	38	31	2A	23	1C	15	E	7
-6	A	0	FA	F4	EE	E8	E2	DC	D6	30	2A	24	**1E**	18	12	C	6
-5	B	0	FB	F6	F1	EC	E7	E2	DD	28	23	**1E**	19	14	F	A	5
-4	C	0	FC	F8	F4	F0	EC	E8	E4	20	1C	18	14	10	C	8	4
-3	D	0	FD	FA	F7	F4	**F1**	EE	EB	18	15	12	F	C	9	6	3
-2	E	0	FE	FC	FA	F8	F6	F4	F2	10	E	C	A	8	6	4	2
-1	F	0	FF	FE	FD	FC	FB	FA	F9	8	7	6	5	4	3	2	1

Таблица А,4. Беззнаковое умножение

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	**1E**
3	0	3	6	9	C	F	12	15	18	**1B**	**1E**	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	**1E**	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	**1E**	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	**1B**	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	**1E**	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	**46**	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	**1E**	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Таблицы А.5 и А.6 содержат результат деления с отсечением. В табл. А.5 для случая деления максимального по модулю отрицательного числа на -1 показан результат, равный 8 с переполнением, хотя на большинстве машин такая операция либо будет запрещена, либо ее результат будет неопределенным.

Таблица А.5. Знаковое короткое деление (строка * столбец)

									-8	-7	-6	-5	-4	-3	-2	-1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	-	1	0	0	0	0	0	0	0	0	0	0	0	0	0	F
	2	-	2	1	0	0	0	0	0	0	0	0	0	0	0	F	E
	3	-	3	1	1	0	0	0	0	0	0	0	0	0	F	F	D
	4	-	4	2	1	1	0	0	0	0	0	0	0	F	F	E	C
	5	-	5	2	1	1	1	0	0	0	0	0	F	F	F	E	B
	6	-	6	3	2	1	1	1	0	0	0	F	F	F	E	D	A
	7	-	7	3	2	1	1	1	1	0	F	F	F	F	E	D	9
-8	8	-	8	C	E	E	F	F	F	1	1	1	1	2	2	4	8
-7	9	-	9	D	E	F	F	F	F	0	1	1	1	1	2	3	7
-6	A	-	A	D	E	F	F	F	0	0	0	1	1	1	2	3	6
-5	B	-	B	E	F	F	F	0	0	0	0	0	1	1	1	2	5
-4	C	-	C	E	F	F	0	0	0	0	0	0	0	1	1	2	4
-3	D	-	D	F	F	0	0	0	0	0	0	0	0	0	1	1	3
-2	E	-	E	F	0	0	0	0	0	0	0	0	0	0	0	1	2
-1	F	-	F	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Таблица А.6. Беззнаковое короткое деление (строка ÷ столбец)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	-	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	-	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	-	3	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4	-	4	2	1	1	0	0	0	0	0	0	0	0	0	0	0
5	-	5	2	1	1	1	0	0	0	0	0	0	0	0	0	0
6	-	6	3	2	1	1	1	0	0	0	0	0	0	0	0	0
7	-	7	3	2	1	1	1	1	0	0	0	0	0	0	0	0
8	-	8	4	2	2	1	1	1	1	0	0	0	0	0	0	0
9	-	9	4	3	2	1	1	1	1	1	0	0	0	0	0	0
A	-	A	5	3	2	2	1	1	1	1	1	0	0	0	0	0
B	-	B	5	3	2	2	1	1	1	1	1	1	0	0	0	0
C	-	C	6	4	3	2	2	1	1	1	1	1	1	0	0	0
D	-	D	6	4	3	2	2	1	1	1	1	1	1	1	0	0
E	-	E	7	4	3	2	2	2	1	1	1	1	1	1	1	0
	-	F	7	5	3	3	2	2	1	1	1	1	1	1	1	1

Таблицы А.7 и А.8 содержат остатки при делении с отсечением. В табл. А.7 для случая деления максимального по модулю отрицательного числа на -1 показан результат, равный 0 с переполнением, хотя на большинстве машин такая операция либо будет запрещена, либо ее результат будет неопределенным.

Таблица АЛ. Остаток при знаковом коротком делении (строка ÷ столбец)

		0	1	2	3	4	5	6	7	-8 8	-7 9	-6 A	-5 B	-4 C	-3 D	-2 E	-1 F
	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	-	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
	2	-	0	0	2	2	2	2	2	2	2	2	2	2	2	0	0
	3	-	0	1	0	3	3	3	3	3	3	3	3	3	0	1	0
	4	-	0	0	1	0	4	4	4	4	4	4	4	0	1	0	0
	5	-	0	1	2	1	0	5	5	5	5	5	0	1	2	1	0
	6	-	0	0	0	2	1	0	6	6	6	0	1	2	0	0	0
	7	-	0	1	1	3	2	1	0	7	0	1	2	3	1	1	0
-8	8	-	0	0	E	0	D	E	F	0	F	E	D	0	E	0	0
-7	9	-	0	F	F	D	E	F	0	9	0	F	E	D	F	F	0
-6	A	-	0	0	0	E	F	0	A	A	A	0	F	E	0	0	0
-5	B	-	0	F	E	F	0	B	B	B	B	B	0	F	E	F	0
-4	C	-	0	0	F	0	C	C	C	C	C	C	C	0	F	0	0
-3	D	-	0	F	0	D	D	D	D	D	D	D	D	D	0	F	0
-2	E	-	0	0	E	E	E	E	E	E	E	E	E	E	E	0	0
-1	F	-	0	F	F	F	F	F	F	F	F	F	F	F	F	F	0

Таблица А.8. Остаток при беззнаковом коротком делении (строка ÷ столбец)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	-	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	-	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2
3	-	0	1	0	3	3	3	3	3	3	3	3	3	3	3	3
4	-	0	0	1	0	4	4	4	4	4	4	4	4	4	4	4
5	-	0	1	2	1	0	5	5	5	5	5	5	5	5	5	5
6	-	0	0	0	2	1	0	6	6	6	6	6	6	6	6	6
7	-	0	1	1	3	2	1	0	7	7	7	7	7	7	7	7
8	-	0	0	2	0	3	2	1	0	8	8	8	8	8	8	8
9	-	0	1	0	1	4	3	2	1	0	9	9	9	9	9	9
A	-	0	0	1	2	0	4	3	2	1	0	A	A	A	A	A
B	-	0	1	2	3	1	5	4	3	2	1	0	B	B	B	B
C	-	0	0	0	0	2	0	5	4	3	2	1	0	C	C	C
D	-	0	1	1	1	3	1	6	5	4	3	2	1	0	D	D
E	-	0	0	2	2	4	2	0	6	5	4	3	2	1	0	E
F	-	0	1	0	3	0	3	1	7	6	5	4	3	2	1	0

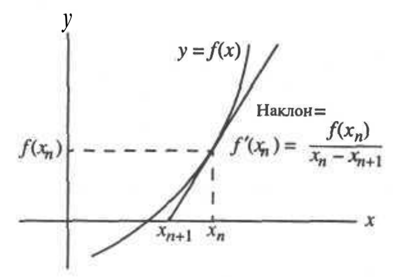
ПРИЛОЖЕНИЕ Б

МЕТОД НЬЮТОНА

Для краткого рассмотрения метода Ньютона предположим, что у нас имеется дифференцируемая функция f от действительной переменной x и нам требуется решить уравнение $f(x)=0$ относительно x. Метод Ньютона по данному приближению x_n корня f дает нам при определенных условиях улучшенное приближение x_{n+1} по формуле

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}.$$

Здесь $f'(x_n)$ означает производную f в точке $x=x_n$. Вывод этой формулы можно пояснить приведенным ниже рисунком (решая показанное уравнение относительно x_{n+1}).



Этот метод хорошо работает для простых доброкачественных функций, например полиномов, если первое приближение достаточно близко к решению. Если приближенное значение достаточно близко к точному, метод обладает квадратичной сходимостью. Это означает, что если r — точное значение корня, а x_n — достаточно близкая оценка, то

$$|x_{n+1}-r|\le(x_n-r)^2.$$

Таким образом, количество точных цифр решения удваивается при каждой итерации (например, если $|x_n-r|<0.001$, то $|x_{n+1}-r|\le 0.000001$).

Если первое приближение далеко от значения корня, то это может привести к медленной сходимости итераций, расходимости в бесконечность, сходимости к другому корню (не являющемуся ближайшим к первому приближению) или привести к бесконечному циклу с некоторым набором значений.

Приведенное рассмотрение метода достаточно туманно из-за наличия фраз типа "определенные условия", "доброкачественная функция" и "достаточно близко". Для того чтобы познакомиться с методом Ньютона более строго, обратитесь практически к любому учебнику по вычислительным методам.

Несмотря на предупреждения об области применения данного метода, зачастую он очень полезен в области целых чисел. Для того чтобы определить применимость этого метода для той или иной функции, вы должны провести соответствующий анализ, наподобие описанного в разделе 11.1.

В табл. Б.1 приведены некоторые итеративные формулы, выведенные из метода Ньютона. В первом столбце приведены искомые значения, во втором — функции, для которых эти значения являются корнями, и в третьем — правые части формул Ньютона, соответствующие этим функциям.

Таблица Б.1. Метод Ньютона для вычисления некоторых значений

Вычисляемое значение	Функция	Итеративная формула
$\sqrt{a}$	$x^2 - a$	$\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$
$\sqrt[3]{a}$	$x^3 - a$	$\frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right)$
$\frac{1}{\sqrt{a}}$	$x^{-2} - a$	$\frac{x_n}{2}\left(3 - ax_n^2\right)$
$\frac{1}{a}$	$x^{-1} - a$	$x_n\left(2 - ax_n\right)$
$\log_2 a$	$2^x - a$	$x_n + \frac{1}{\ln 2}\left(\frac{a}{2^{x_n}} - 1\right)$

К сожалению, не всегда просто найти подходящую функцию. Разумеется, существует множество функций, корнем которых является интересующее нас значение, но только немногие из них приводят к практичным итеративным формулам. Обычно функция представляет собой некоторый тип обратных вычислений по отношению к вычислению искомого значения. Например, для поиска $\sqrt{a}$ используется $f(x) = x^{-} - a$, для поиска $\log_2 a$ — $f(x) = 2^x - a$ и т.д.[1]

Итеративная формула для $\log_2 a$ сходится (к $\log_2$ a) даже если заменить множитель $1/\ln 2$ некоторым другим значением (например, 1 или 2). Однако скорость сходимости при этом упадет. Для ряда приложений вместо множителя $1/\ln 2$ можно использовать величины 3/2 или 23/16 ($1/\ln 2 \approx 1.4427$).

[1] Метод Ньютона для частного случая квадратного корня был известен еще в древнем Вавилоне около 4000 лет назад.

ИСТОЧНИКИ ИНФОРМАЦИИ

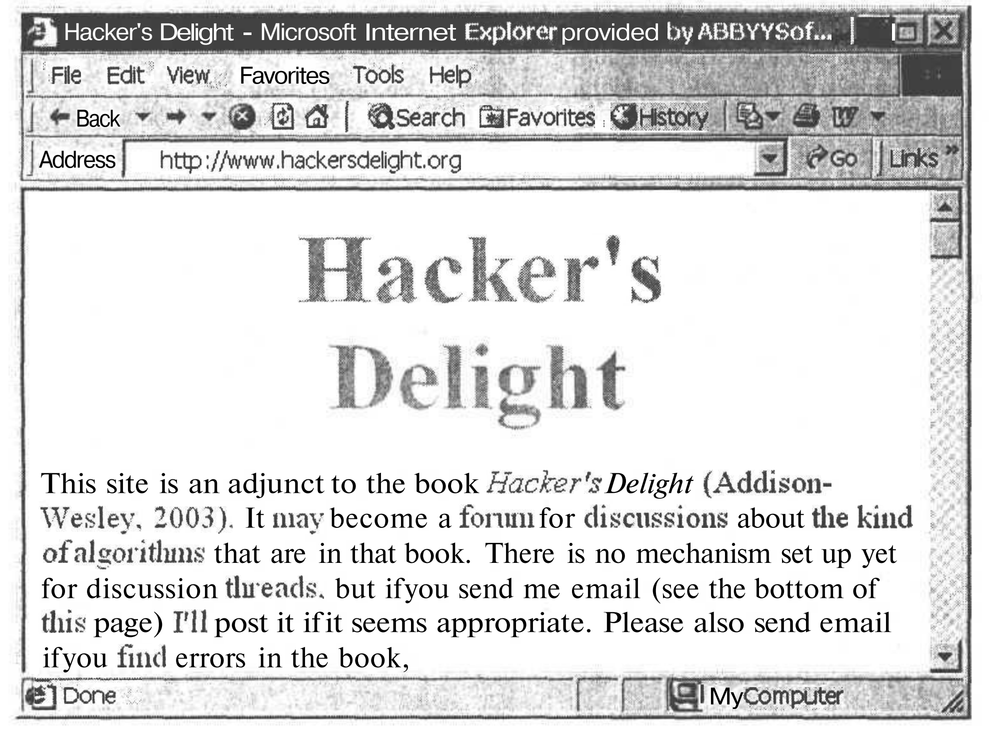
1. *Advanced Encryption Standard (AES)*. — National Institute of Standards and Technology, FIPS PUB 197 (November 2001). Доступно по адресу: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.
2. Alverson R. Integer Division Using Reciprocals. — In: *Proceedings IEEE 10th Symposium on Computer Arithmetic*. — Grenoble, France. — June 26–28, 1991. — P. 186–190.
3. Найдено в подпрограмме интерпретатора REXX, написанной Марком Аусландером (Marc A. Auslander).
4. Auslander M. A. Частное сообщение.
5. Bernstein R. Multiplication by Integer Constants // *Software — Practice and Experience*. — 1986. — 16, 7. — P. 641–652.
6. Burks A. W., Goldstine H. H., and von Neumann J. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, Second Edition (1947). — In: *Papers of John von Neumann on Computing and Computing Theory* (vol. 12 in the Charles Babbage Institute Reprint Series for the History of Computing). — MIT Press, 1987.
7. Stephenson C. J. Частное сообщение.
8. На эти правила указал Норман Коген (Norman H. Cohen).
9. Cutland N. J. *Computability: An Introduction to Recursive Function Theory*. — Cambridge University Press, 1980.
10. Hoxey, Karim, Hay, and Warren (Ed.). *The PowerPC Compiler Writer's Guide*. — Warthman Associates, 1996.
11. *Data Encryption Standard (DES)*. — National Institute of Standards and Technology, FTPS PUB 46-2 (December 1993). Доступно по адресу: `http://www.itl.nist.gov/fipspubs/fip46-2.htm`.
12. Dewdney A. K. *The Turing Omnibus*. — Computer Science Press, 1989.
13. Dudley, Underwood. History of a Formula for Primes // *American Mathematics Monthly 76*. — 1969. — P. 23-28.
14. Ercegovac M. D. and Lang T. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. — Kluwer Academic Publishers, 1994.
15. Gardner M. Mathematical Games // *Scientific American*. — 1972. — 227, 2. — P. 106–109.
16. Gregoire D. G., Groves R. D., and Schmookler M. S. *Single Cycle Merge/Logic Unit*. — US Patent No. 4,903,228. — 1990.
17. Granlund T. and Kenner R. Eliminating Branches Using a Superoptimizer and the GNU C Compiler. — In: *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), July 1992. — P. 341-352.
18. Graham R. L., Knuth D. E., and Patashnik O. *Concrete Mathematics: A Foundation for Computer Science, Second Edition*. — Addison-Wesley, 1994. (Перевод: Грэхем Р., Кнут Д., Паташник О. *Конкретная математика. Основание информатики*. — М.: Мир, 1998.)
19. Steele G. L., Jr. Частное сообщение.
20. Steele G. L., Jr. Arithmetic Shifting Considered Harmful. — AI Memo 378, MIT Artificial Intelligence Laboratory (September 1976); also: *SIGPLAN Notices 12*. — 1977. — 11. — P. 61–69.
21. Granlund T. and Montgomery P. L. Division by Invariant Integers Using Multiplication. — In: *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (PLDI). — August 1994. — P. 61-72.

22. Второе выражение предложено Ричардом Голдбергом (Richard Goldberg).
23. Goodstein, Prof. R. L. Formulae for Primes // *The Mathematical Gazette.* — 1967. — 51. — P. 35-36.
24. Обнаружено в GNU Superoptimizer.
25. Beeler M., Gosper R. W., and Schroeppel R. *HAKMEM.*— MIT Artificial Intelligence Laboratory AIM 239. — February 1972.
26. Hay R. W. Частное сообщение.
27. Первое выражение найдено в подпрограмме, написанной Р. Хеем (R. W. Hay).
28. Hilbert D. Ueber die stetige Abbildung einer Linie auf ein Flächenstück // *Mathematischen Annalen.* — 1891. — 38. — P. 459–460.
29. Hopkins M. E. Частное сообщение.
30. Hillis W. D. and Steele G. L., Jr. Data Parallel Algorithms // *Comm. ACM 29.* — 1986. — 12. — P. 1170–1183.
31. Hennessy J. L. and Patterson D. A. *Computer Architecture: A Quantitative Approach.* — Morgan Kaufmann, 1990.
32. Harbison S. P. and Steele G. L., Jr. *C: A Reference Manual,* Fourth Edition. — Prentice-Hall, 1995.
33. Hardy G. H. and Wright E. M. *An Introduction to the Theory of Numbers,* Fourth Edition. — Oxford University Press, 1960.
34. Из курса программирования IBM, 1961.
35. Irvine M. M. Early Digital Computers at Bell Telephone Laboratories // *IEEE Annals of the History of Computing 23.* — 2001. — 3. — P. 22–42.
36. Von Neumann J. First Draft of a Report on the EDVAC. — In: *Papers of John von Neumann on Computing and Computing Theory* (vol. 12 in the Charles Babbage Institute Reprint Series for the History of Computing). — MIT Press, 1987.
37. Найдено в компиляторе GNU C, перенесенном на платформу IBM RS/6000 Ричардом Кеннером (Richard Kenner).
38. Knuth D. E. *The Art of Computer Programming. Vol. 1. Fundamental Algorithms, Third Edition.* — Addison-Wesley, 1997. (Перевод: Кнут Д.Э. *Искусство программирования. Том 1. Основные алгоритмы, 3-е изд.* — М. : Издат. дом "Вильяме", 2000.)
39. Knuth D. E. *The Art of Computer Programming. Vol. 2. Seminumerical Algorithms, Third Edition.* — Addison-Wesley, 1998. (Перевод: Кнут Д.Э. *Искусство программирования. Том 2. Получисленные алгоритмы, 3-е изд.* — М. : Издат. дом "Вильяме", 2000.)
40. Идея использовать в качестве основания системы счисления отрицательные числа была независимо открыта различными людьми. Наиболее раннее упоминание этой идеи, по Кнуту, — в 1885году Витторио Грюнвальдом (Vittorio Grünwald). Более детально этот вопрос рассмотрен во втором томе *Искусства программирования* Кнута [39].
41. Kruskal C. P., Rudolph L., and Snir M. The Power of Parallel Prefix // *IEEE Transactions on Computers C-34.* — 1985. — 10. — P. 965-968.
42. Lamport L. Multiple Byte Processing with Full-Word Instructions // *Communications of the ACM 18.* — 1975. — 8. — P. 471–475.
43. Lee R. B., Shi Z., and Yang X. Efficient Permutation Instructions for Fast Software Cryptography // *IEEE Micro 21.* — 2001. — 6. — P. 56–69.
44. Lam W. M. and Shapiro J. M. A Class of Fast Algorithms for the Peano-Hilbert Space-Filling Curve. — In: *Proceedings ICIP 94.* — 1994. — 1. — P. 638–641.
45. Denneau M. Частное сообщение.
46. Kane G. and Heinrich J. *MIPS RISC Architecture.* — Prentice-Hall, 1992.

47. Morton M. Quibbles & Bits // *Computer Language 7.* — 1990. — 12. — P. 45-55.
48. В будущем издании *Искусства программирования*. Доступно по адресу: `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`.
49. Niven I., Zuckerman H. S., and Montgomery H. L. *An Introduction to the Theory of Numbers,* Fifth Edition. — John Wiley & Sons, Inc., 1991.
50. Purdom P. W., Jr., and Brown C. A. *The Analysis of Algorithms.* — Holt, Rinehart and Winston, 1985.
51. Oden P. H. Частное сообщение.
52. Я нашел этот прием в компиляторе PL.8.
53. Ribenboim P. *The Little Book of Big Primes.* — Springer-Verlag, 1991.
54. Reingold E. M., Nievergelt J., and Deo N. *Combinatorial Algorithms: Theory and Practice.* — Prentice-Hall, 1977.
55. Sagan H. *Space-Filling Curves.* — Springer-Verlag, 1994[1].
56. Shepherd A. D. Частное сообщение.
57. Stallman R. M. *Using and Porting GNU CC.* — Free Software Foundation, 1998.
58. Voorhies D. Space-Filling Curves and a Measure of Coherence. — In: *Graphics Gems II,* AP Professional, 1991.
59. Warren H. S., Jr. Functions Realizable with Word-Parallel Logical and Two's-Complement Addition Instructions // *Communications of the ACM 20.* — 1977. — 6. — p. 439–441.
60. Wegner, P. A. A Technique for Counting Ones in a Binary Computer // *Communications of the ACM 3.* — 1960. — 5. — P. 322[2].
61. Wells D. *The Penguin Dictionary of Curious and Interesting Numbers.* — Penguin Books, 1997.
62. Willans C. P. On Formulae for the *n*th Prime Number // *The Mathematical Gazette 48.* — 1964. — P. 413–415.
63. Wormell C. P. Formulae for Primes // *The Mathematical Gazette 51.* — 1967. — P. 36–38.

[1] Отличная книга, которую я рекомендую всем, кого интересует данная тема.
[2] Самое ранее упоминание об этом методе, известное мне.

Не так давно автором данной книги открыт Web-узел `www.HackersDelight.org`, основная цель которого — выступать в роли активного дополнения к книге. На нем планируется разместить материалы, как непосредственно связанные с книгой (например, все исходные тексты фрагментов кода из книги в электронном виде или последние исправления и дополнения, внесенные в книгу), так и материалы, близкие "по духу" к материалу книги.



Так, на момент написания этого дополнения здесь можно было найти ряд интересных ссылок на другие Web-узлы, посвященные подобной тематике. Здесь же имеется программа *A Hacker's Assistant* — оптимизатор, позволяющий находить минимальный код без ветвлений для ряда специальных функций. Представлен также ряд графиков различных дискретных функций из числа рассматривавшихся в книге — графиков, которые стоят того, чтобы посмотреть на них хотя бы ради получения эстетического удовольствия. Это действительно необычное и очень поучительное зрелище.

Здесь же имеются и дополнительные материалы к рассмотренным в книге темам — например, поиск "магических" чисел, использующихся при делении на константы, рассматривается вопрос о вычислении CRC32, материалы, присланные читателями книги (к сожалению, пока что автор не реализовал на своем Web-узле форум для обмена мнениями и информацией).

И последнее — по порядку, но не по важности: именно здесь можно найти материалы, которые автор планирует включить в новое издание книги, так что вы можете оказаться ее первыми читателями еще до того, как книга попадет в типографию!

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

П

Р

С

Т

У

Ф

Ц

Ч

Научно-популярное издание

Генри С. Уоррен

Алгоритмические трюки для программистов
Исправленное издание

Литературный редактор *Т.П. Кайгородова*
Верстка *А.А. Линник, мл.*
Художественный редактор *В.Г. Павлютин*
Корректоры *Л.А. Гордиенко* и *О.В. Мишутина*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 15.12.2003. Формат 70Х100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 23,22. Уч.-изд. л. 17,7.
Тираж 3500 экз. Заказ № 1296.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.